Radio Shack®

TRS-XENIX™ Operations Guide

CUSTOM MANUFACTURED IN U.S.A. BY RADIO SHACK, A DIVISION OF TANDY CORPORATION

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE"

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

IMPORTANT

****
* TRSDOS Application Users: If you are *
* currently running your Radio Shack Application *
* under TRSDOS 2.Øa (or 2.Øb) and you plan to *
* upgrade to TRS-XENIX Applications, you must *
* transfer your data files to TRSDOS-II on your *
* hard disk before upgrading to TRS-XENIX. If *
* you have not initialized your hard disk with *
* TRSDOS-II, follow the instructions in your *
* Hard Disk Owner's Manual. Then follow the *
* instructions given here to transfer your *
* TRSDOS-II files to TRS-XENIX. *
****

If you have been using TRSDOS™-II on your hard disks prior to receiving your TRS-XENIX Runtime System, please note the following:

- TRS-XENIX™ overwrites any information already stored on your hard disk. Carefully follow the instructions given here to save important data and programs for later use.
- Be sure to SAVE your TRSDOS-II programs and data to floppy disks before installing TRS-XENIX.
- Only TRSDOS-II 4.Ø (or later) files can be transferred to TRS-XENIX. If you have any TRSDOS 2.Øa (or 2.Øb) files that you want to transfer to TRS-XENIX, you must first convert them to TRSDOS-II using the FCOPY command.
- You can still run your TRSDOS and TRSDOS-II programs from floppy disk.

The following pages explain how to:

1. FCOPY any TRSDOS 2.Øa (2.Øb) programs and data you wish to save over to your hard disk.
2. SAVE all files on your hard disk to floppy disks.

Note that by FCOPYing your TRSDOS programs to hard disk first, all your files are located on set of floppy disks after the SAVE process.

FCOPY Your TRSDOS Files to TRSDOS II

The FCOPY command copies files from floppy disk TRSDOS 1.2, 1.2a, 2.Ø, 2.Øa, or 2.Øb to a disk formatted by TRSDOS-II (and vice versa except to TRSDOS 1.2 or 1.2a).

(For more information on FCOPY, see your TRSDOS-II Reference Manual.)

1. Power up your computer under TRSDOS-II Version 4.1 (or later).

2. Insert your TRSDOS application disk in Drive Ø.

3. Enter the FCOPY command at TRSDOS-II Ready.

   FCOPY Ø TO 4 {SYS,ALL} <ENTER>

   This copies all files, system and user, that are on the TRSDOS application disk in Drive Ø to Drive 4.

Alternate Method: FCOPYing A File. If you know the filespecs of the programs to be converted, you may want to use a different syntax form:

FCOPY filespec:drive TO drive

This is especially useful when you convert only a few programs.

For example:

FCOPY SAMPLE/DAT:Ø TO 4

copies the file SAMPLE/DAT from the disk in Drive Ø to Drive 4.

FCOPY Your Data Disks to TRSDOS-II

1. Insert the TRSDOS data disk (source disk) into Drive Ø.

2. At TRSDOS-II Ready, type:

   FCOPY Ø TO 4 {SYS,ALL} <ENTER>

3. TRSDOS-II stores your data on Drive 4. When it finishes the conversion, TRSDOS-II displays the message **FCOPY Complete**.

SAVE Your Hard Disk Files to Floppy Disks

You are now ready to save all of the programs and data stored on your hard disk.

> **Important:** Since this is a very important step in preparing to install TRS-XENIX on your hard disk, we strongly suggest making at least **two** SAVE Data Sets.

Be sure to have a supply of blank disks ready. Do not format them; SAVE organizes the disks into its own special format.

Remember, these diskettes will be your archive file of all of your programs that were stored on your hard disk under TRSDOS-II. Take the time to put the date, volume and data set numbers on each diskette. You may also want to put descriptions on each diskette.

To save all files from your hard disk (programs, data, and system files), enter the following command at TRSDOS-II Ready:

```
SAVE :4 :Ø {SYS,ALL,ABS}  <ENTER>
```

If you have secondary hard disks, enter the following command for each:

```
SAVE :d :Ø {ALL,ABS}  <ENTER>
```

where d is the hard disk from which the files are being SAVEd. The files are stored on disks in Drive Ø. As one disk becomes full, TRSDOS-II prompts you for the next disk.

When all of the files are saved, TRSDOS-II prompts you to insert Volume Ø of your Save Data Set into Drive Ø. TRSDOS-II now updates that disk with housekeeping information.

When the save is finished, you are returned to TRSDOS-II Ready. Repeat the procedure again to create a Backup Save Data Set.

TRS-80®

TRS-XENIX OPERATIONS GUIDE

Radio Shack®

Notes to Users of TRS-XENIX

- This manual does not attempt to discuss every command and option included on your TRS-XENIX Runtime System Diskette. Only those commands that are necessary to effectively run applications under TRS-XENIX are described.

- The RUNCOBOL program is included with TRS-XENIX Runtime so you can run your TRS-XENIX application programs.

- When running TRS-XENIX, the hard disks are numbered Ø-3, where Drive Ø is the primary hard disk.

- Do not turn on the write-protect switch on your hard disks or remove the write-enable tab from your floppy disks when running TRS-XENIX. Approximately every 3Ø seconds TRS-XENIX will access the drives to update its files and directories.

- TRS-XENIX uses the "Media Error Map" on the bottom of your hard disk to format the disk. If the map for your hard disk is missing, contact your Radio Shack dealer for assistance.

- The **mail** program is included in the TRS-XENIX Software Development Package (26-64Ø1). Commands mentioned in this guide which use **mail** will operate normally, although the mail operation will not be invoked.

- TRS-XENIX Applications user's should look at Appendix E for instructions on installing their applications. Making backups of both the programs and the data are also described.

CONTENTS

Appendixes

CHAPTER 1

Introduction

This Operations Guide is designed to help you learn the basics of your new TRS-XENIX system, and to help you get the system up and running on your computer. Even if you have never used TRS-XENIX, this guide gives you enough information to give you confidence in your role as system manager or as a user.

This chapter defines the role of the system manager and describes what you get with your TRS-XENIX system. Since the system manager's job is a crucial one with many aspects, you should learn as much as you can about TRS-XENIX.

Each section provides directions for carrying out the procedures for which you will be responsible, along with an overview of TRS-XENIX system concepts. These procedures have been designed to make your job as easy as possible. If you read the instructions carefully, you should have little difficulty in starting to use your TRS-XENIX system.

By the time you have finished this guide and feel comfortable with the tasks described, you should be familiar with a number of basic TRS-XENIX commands. The following are some important items you are introduced to in this guide:

- The TRS-XENIX file structure -- the way in which the TRS-XENIX programs and user programs and data are organized.
- Editor -- you should become familiar with **ed**, a TRS-XENIX text editor. With **ed** you can make additions and changes to files.
- The Shell -- one of TRS-XENIX's tools for increasing your productivity on the system.

To do this, there is no substitute for reading whatever TRS-XENIX documentation you have available and wherever possible, trying out the examples given in the tutorials.

1.1 The Role of the System Manager

You may have heard the term "system manager" or "system administrator" used to describe the person with the overall responsibility for the care and maintenance of a computer system. You should appoint someone -- either yourself or someone reliable -- to the position of system manager. On a large computer, this is often a full time job for one or more people. On your system, of course, you may have only a few users or maybe just yourself. However, this makes the system manager's job no less critical. Your tasks will include:

- the initial installation of the TRS-XENIX system
- adding and removing user accounts, passwords, and file systems
- ensuring that system resources (time and disk space) are efficiently distributed among the users
- "backing up" or making copies of all files on the system to guard against the loss of programs and data, in case of user or hardware errors.

You will also need to manage a library of floppy disks and other storage media containing system backups, user files and application programs. This minimizes the chance of potentially time-consuming and expensive losses of programs, data, and text.

Sometimes your job may make you unpopular with your users. For example, persuading users to remove their little-used files from an overcrowded disk can be frustrating. On the other hand, doing your job well will result in an efficient, productive system with efficient and productive (and satisfied) users.

> **HINT:** The "system manager" logs in as the "root" of the TRS-XENIX system. The root is the most powerful user in TRS-XENIX. He is also called the "super-user".

1.2 The TRS-XENIX System

Like any other operating system, TRS-XENIX is a collection of programs which are resident in a computer at all times and are designed to control its resources. These programs control:

- communication between the central processor, input/output devices (line printers, terminals), and storage devices (hard and floppy disk drives).
- the user's access to files on the system
- processing time and disk space, ensuring that both are fairly distributed among users.
- provides a way for user programs to communicate with the physical components of the computer, or hardware.

1.3 A Word On TRS-XENIX

TRS-XENIX is derived from UNIX, an operating system developed by Bell Labs a number of years ago and widely used on larger computer systems. TRS-XENIX is a multi-user and multi-tasking system. That is, it allows more than one user simultaneous access to its resources, and allows more than one process, or job, to run at the same time.

In most respects, the TRS-XENIX on your system is identical to that found on larger and more expensive systems. However, the extent to which you can utilize its features depends on the size and characteristics of your computer.

Naturally, several users cannot work simultaneously unless you have terminals attached to your system. However with TRS-XENIX you can still create separate directories and file systems for different users to access at different times.

CHAPTER 2

Before You Start

If you are using TRS-XENIX for the first time, you may feel some apprehension. Therefore, this chapter gives some basic information to make you feel more comfortable about its operation. You should read this chapter before beginning to install and use TRS-XENIX. It explains:

- what the various keys on the keyboard do
- TRS-XENIX file and directory naming conventions
- how to use TRS-XENIX commands
- how to start and stop your system safely and what to do if the system halts accidentally

2.1 Your Keyboard

In most respects, the keyboard of your terminal is exactly like that of a typewriter. Most of the letters, numerals, and punctuation marks are in the same place, and you will quickly discover that the "spacebar", "backspace", "repeat", and "shift" keys behave in much the same way as they would on any typewriter. However, there are a few differences you should be aware of before you begin working.

You may not use keys that "look alike" interchangeably -- for example the upper-case letter "O" and the number Ø (zero), or the lower-case "l" and the number 1 (one). All computers will recognize these as separate characters, so be sure you always type the correct one.

Some of the keys on your keyboard have a special meaning when you are using TRS-XENIX. These include control keys or sequences used to produce special TRS-XENIX characters that don't appear on your keyboard. They also produce various "escape" sequences which are used to exit from programs, terminate activities, log out, or stop the movement of text on your screen.

Angle brackets (< >) are used in this guide to represent keys. Note that whenever you are asked to type the "Control Key" (<CTRL>) along with some other key, you should always press the <CTRL> key first, and hold it down while you type the second key, just as you would hold down the shift key on a typewriter while typing the letter you want to capitalize. Here is a list of the most commonly used characters and sequences.

<CTRL><S>	Stops text from "scrolling", that is, moving up and off the screen. To start scrolling again, type <CTRL><Q> The <HOLD> key may be used from the console.
<CTRL><D>	You will have several important uses for the <CTRL><D> sequence: - to log out - to bring the system up from maintenance mode
<BREAK>	In addition to special uses it may have in some TRS-XENIX programs, the <BREAK> key will interrupt any command and return you to the system prompt ($).
<ENTER>	Type this key after a command for TRS-XENIX to receive the instructions. In some documentation you may also see this key named <RETURN>, for "carriage return".
backslash (\)	Used in some advanced features of TRS-XENIX. The TRS-XENIX backslash (\) character is obtained by typing <CTRL></> from the console.
pipe (|)	Used in some advanced features of TRS-XENIX. It can be entered by typing <CTRL><1> (one) from the console.

2.2 TRS-XENIX File and Directory Naming Conventions

All TRS-XENIX files and directories may have names up to fourteen characters long, including any combination of upper- or lower-case letters, numbers, and the period (.). You may not use the slash (/) character, or any other punctuation or special characters because they often have special meanings for TRS-XENIX (See Chapter 1Ø).

Be especially cautious in your use of the two "wild card" characters, question mark (?) and asterisk (*). Wild card characters are used to replace file names or parts of file names. The "?" replaces a single character and the "*" replaces several characters, or even an entire name.

For example, to save typing you might refer to your file chapt2.s as any of the following:

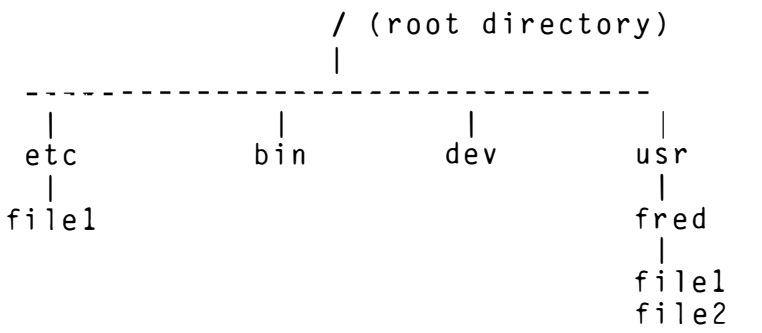
```
chap?.s          *2.s          *2*
```

If you want to use a command involving multiple files, you can use wild carding to process all of them at once. So, if you want to refer to all of your chapters (e.g., chapl.s, chap2.s,etc.), you can type:

```
*.s
```

to tell TRS-XENIX that you mean "all the files ending in ".s". Using the "*" by itself means "all files". Of course, if you are using a TRS-XENIX command like "remove files", you should be extremely careful about using wild card characters.

TRS-XENIX files are grouped in directories and arranged hierarchically. That is, a directory, which contains a collection of files, may be a member of yet another directory. This would resemble a tree with branches:



The root of this tree is called the "root directory", and is denoted by the "slash" (/).

TRS-XENIX identifies files by their "pathnames", the path you take along the branches of this tree to arrive at the desired file. The steps along the way are separated by slashes (/). In the diagram, for example, there are two unique files called file1. One is /etc/file1 and the other is /usr/fred/file1.

The TRS-XENIX file structure is discussed at much greater length in Chapter 5. For now, however, it is sufficient that you understand the "pathname" concept when working with directories and files.

2.3 How to Use TRS-XENIX Commands

A TRS-XENIX command may be typed whenever the prompt appears, indicating that you are in the "shell", the command level of the system. This prompt will either be a dollar sign ($) or a number sign (#). TRS-XENIX commands generally consist of single lower-case words, which may be followed by one or more "switches" and "arguments". Most often the argument is a file name. Every command must be followed by typing <ENTER>.

To take a simple example, you can use the command **cat** to look at the contents of a file. If you type:

```
cat file1 <ENTER>
```

the contents of file1 are displayed on your screen.

To use another example, the command **lc** lists the contents of a specified directory. Therefore, the command line:

lc /usr/fred <ENTER>

gives you a list of files and directories in the directory /usr/fred.

> **HINT:** When you type TRS-XENIX commands, be sure to include or omit spaces exactly as indicated; it is best to think of the "space" as a character like any other.

2.4 Stopping the System Safely

You should carefully read and follow the directions in Chapter Three for the correct TRS-XENIX shutdown procedure.

You should **NOT** press the RESET or POWER switches, turn the hard disk off, or unplug your computer while TRS-XENIX is running.

If you do accidentally stop the system, or experience a power failure or other disaster, the next time you attempt to start, or "boot" the system, TRS-XENIX will warn you that the last shutdown was not done correctly. You will be asked whether you wish to proceed with the "cleaning" of the file system. Always answer <y>. TRS-XENIX will then use a five-phase system-checking program called **fsck** to check for system inconsistencies and repair them.

CHAPTER 3

Starting Up

Before you can use TRS-XENIX you will have to format the hard disk, and transfer the information from the floppy disks in your distribution package to the hard disk. Ordinarily, you will not have to repeat this installation process, but you will need to save your distribution disks in case of the accidental destruction of the system on the disk in which case, you can reconstruct your system by repeating the installation procedure.

Allow plenty of time to complete the installation. The formatting phase of the installation alone takes approximately twenty minutes, depending on the size of your hard disk. The initialization of the system and the transfer of all TRS-XENIX system files to hard disk takes at least a half hour. Also, although the installation programs are designed to guide you through the process step-by-step, you should take the time to read over the description of the procedures before you begin.

3.1 Installing TRS-XENIX on the Hard Disk

Briefly, the installation of TRS-XENIX on a hard disk system consists of the following phases:

- "Boot", or start the system running under floppy disk control, with the TRS-XENIX **Boot Disk**. Run **diskutil** to format the hard disk.
- Boot TRS-XENIX again under floppy disk control using the same **Boot Disk** and copy the contents of the floppy disk to the hard disk.
- Boot TRS-XENIX from the hard disk and copy the rest of TRS-XENIX from the **Install Disks** to the hard disk.

Read the instructions carefully before you begin and follow the procedures exactly as they are written. If you make a mistake at any time during the installation process, you should press the RESET button and repeat the procedures from the beginning.

3.1.1 STEP ONE: The diskutil Program

Diskutil is actually a "stand alone" program which runs **instead** of TRS-XENIX, so you can do some essential tasks before you have brought your TRS-XENIX system up. Besides the initial formatting of the hard disk, this program also enables you to perform three other essential functions:

- format floppy disks
- copy the contents of one floppy disk to another
- format additional hard disks

The procedure for formatting and copying floppy disks is described in detail in Chapter Eight.

To make your work easier, the **diskutil** program will tell you what it is doing and prompt you for answers to some simple questions. During the installation process **diskutil** performs two basic functions:

- formatting the hard disk
- entering the data from the "media error map"

First, you will need to boot the **diskutil** program from the floppy disk labeled "**Boot Disk**".

Important: Before turning on your computer system, carefully remove the **"MEDIA ERROR MAP"** located in the plastic envelope on the underside of the hard disk drive.

Follow these steps:

1) Turn on your computer and all peripherals. Press the <BREAK> key repeatedly until the "Insert Diskette" message is displayed.

2) Insert the **Boot Disk** into floppy Drive Ø.

3) When the boot message appears, type **diskutil** after the colon (:) and press the <ENTER> key:

```
TRS-XENIX Boot
: diskutil <ENTER>
```

4) Answer the following prompts:

```
Diskutil: hard or floppy disk (h or f)?
   Type <h> to format your hard disk.
Copy or format (c or f)?
   Answer <f> to format.
Hard disk unit number  (Ø...3)?
   Answer <Ø> to indicate the first hard disk.
```

5) The next display tells you to look in your Hard Disk Owner's Manual for the next information:

```
How many cylinders?
  (For example 256 for Eight Meg or 23Ø for Twelve
  Meg hard disks)
How many heads?
   (Enter 4 for Eight Meg Drives or 6 for Twelve
   Meg hard disks)
```

6) Using the Media Error Map, type in the list of **TRACK** and **HEAD** numbers as follows. For instance, if the list shows:

TRACK	HEAD	BYTE COUNT	LENGTH
133	ØØ	Ø1333	Ø2
174	Ø1	Ø9826	Ø5

you should type:

```
133,Ø <ENTER>
174,1 <ENTER>
done <ENTER>
```

where directed. If the list is blank, just type

"done" <ENTER>

The screen will look like this:

```
enter numbers or "done": 133,Ø      (first set)
enter numbers or "done": 174,1      (second set)
enter numbers or "done": done       (finished)
```

Of course, you will be entering the numbers from your own Media Error Map instead.

Note: If necessary you may type <BREAK> to abort at any time during this process.

7) The system displays the message:

```
About to format hard disk drive Ø.
```

It also displays a message telling how long the formatting process will take. You may abort the formatting process at any time, but you will be unable to successfully use the drive until the process is completed.

You will see the number of the cylinder and heads change as they are completed:

```
Formatting cylinder nnn, head nn
```

8) When the formatting is completed, you will see the message:

```
Hard disk drive successfully formatted.
Drive parameters and MEDIA ERROR MAP successfully
written. Your hard disk is ready for the TRS-XENIX
initialization.
```

Proceed to, STEP TWO.

3.1.2 STEP TWO: The hdinit Program

The second step in the installation procedure is running a program called **hdinit**. **Hdinit** accomplishes four tasks:

- copies the boot track to hard disk
- creates a TRS-XENIX file system on the hard disk
- copies the contents of the **Boot Disk** to the hard disk
- shuts itself down

Follow these steps to run the **hdinit** program:

9) Press the RESET button, then press the <BREAK> key repeatedly until the boot prompt appears. When the prompt appears press <ENTER>:

```
TRS-XENIX
:  <ENTER>
```

10) The system will respond as if you had typed the word "xenix" after the colon. You should see a message in a box which tells you that your are running TRS-XENIX from the "Installation Floppy".

11) The system will ask you:

Do you wish to initialize your hard disk?
Answer <y> for yes.

Has your hard disk been formatted with diskutil?
Naturally, if you have completed the formatting procedure described earlier, your answer is <y>.

12) Once again, you will be asked for the number of cylinders and heads on your hard disk. Respond with the same numbers you indicated during the formatting of the hard disk with diskutil. (e.g. 256 cylinders and 4 heads for an eight megabyte hard disk.)

13) The system will proceed with the four steps outlined earlier -- installing the boot track, making a file system, copying TRS-XENIX files from the floppy disk to the hard disk, and finally shutting the system down. Do not touch the system until you see the message:

Normal System Shutdown

Note: If for some reason you are repeating this procedure, you may be warned, while the file system is being created, that "mkfs contains data". You will be asked whether to "overwrite". Answer <y> to finish installing the system.

3.1.3 STEP THREE: The firsttime Program

In the final phase of the installation, you will be booting TRS-XENIX from the hard disk and copying the contents of the remaining TRS-XENIX floppy disks (**Install 1** and **Install 2**) to the hard disk:

14) Reboot the system, this time from hard disk (Do **not** press <BREAK>). The system will boot from the hard disk.

15) Once again, you will respond to the boot prompt by typing <ENTER> after the colon (:). This time the message in the box should read:

TRS-XENIX Hard Disk
Basic System
File System Installation.

16) Instruction for inserting and removing your floppy disks are displayed.

17) You are prompted with the question:

First Floppy?:

Insert the **INSTALL 1** floppy disk in Drive Ø; press <y> and then <ENTER>

18) The system tells you that it is:

Extracting files from floppy...

When this is complete, you are prompted with:

Next floppy?

19) Insert the INSTALL 2 disk in Drive Ø, and answer <y> and <ENTER>. The system continues to copy files to the hard disk.

2Ø) Since you have only two floppy disks to "install", answer <n> for no when it asks for the next floppy. You will see the message:

Setting up directories and permissions.
Installation complete.

Your hard disk now contains a complete TRS-XENIX system, and the installation procedure is complete. The next message will be:

Type control-d to proceed with normal startup
(or give root password for system maintenance):

This is the normal boot you will see every time you start your TRS-XENIX system. It is described in the next section.

CHAPTER 3

3.2 Booting Your TRS-XENIX System

The term "boot" is used in this guide to describe the process of bringing TRS-XENIX up on your system, and to distinguish this from powering up the computer, or installing TRS-XENIX on your hard disk for the first time. To boot TRS-XENIX turn on your computer and hard disk. If your system is already on, just flip the RESET switch. The screen clears and then the screen reads:

TRS-XENIX Boot
:

You need only press the <ENTER> key; your system will automatically respond as if you had typed the word "xenix" after the colon (:).

As noted in the preceding chapter, if your system is not shutdown correctly, the next time you attempt to boot the system TRS-XENIX will warn you of that fact. You should always respond <y> when TRS-XENIX asks you whether to proceed with the cleaning of the file system. After TRS-XENIX has finished running the five-phase system-checking program called **fsck** and repaired system inconsistencies, the boot process should continue normally. However, under certain error conditions discovered by **fsck**, the system may shut itself down and ask you to reboot. Repeat the normal boot procedure.

3.3 Login

When you have successfully booted TRS-XENIX, information about the size and release number of your system is displayed. This is followed by the instruction:

```
Type control-d to proceed with normal startup
(or give root password for system maintenance):
```

You should type <CTRL><D> at this point to bring up multi-user TRS-XENIX for yourself and other users.

If you respond with the root password at this point, you will be positioned in "system maintenance" mode (single-user). In system maintenance mode, the functions of TRS-XENIX are limited (described in Chapter 7). In this mode, even the system manager can perform only limited tasks. Be sure you type <CTRL><D>, unless you have a specific reason for being in system maintenance mode.

> **HINT:** Backups, installs, and creating file systems on floppy disks should be done from the system maintenance mode. However, the **save** and **install** commands described in Appendix E must be done from root.

HINT You should change the root password as soon as possible to limit access to the root/system maintenance mode. The root password is initially set to <ENTER>. This is described in the next chapter.

After you type <CTRL><D>, you will be asked for the time. Although the system will accept <ENTER> in lieu of typing the date, you are encouraged to give the system the full date. The accuracy of the dates and times stamped on all the files created and changed on TRS-XENIX depends on the time you enter. In actuality, the year (yy), month (mm), day (dd), and seconds (ss) are all optional; only the hour (hh) and minutes (mm) are required.

HINT You can change the timezone that is displayed by typing **timezone zone <ENTER>**. zone should be a three letter abbreviation such as CST, CDT, MST, MDT, etc. This must be entered from root.

After you enter the time, you will see the single word:

login:

If you are acting as system manager using your system for the first time, type the word "root" and <ENTER>.

Among your first tasks as system manager is the creation of the necessary login information for yourself and all other users of your system. Once these have been established, users may type their own login names, followed by their passwords, to begin working on the system. Note that the password will not appear on the screen as you type it. These procedures are discussed in detail in the next chapter.

Once you have "logged in", the system will respond with a welcoming message, and you will see the TRS-XENIX prompt. If you are logged in as "root", this prompt will be a number sign (#). If you are logged in as a ordinary user, the prompt will be a dollar sign ($). You are now ready to begin using your TRS-XENIX system.

3.4 Halting the System

One last thing you will undoubtedly want to know before going any further is how to stop TRS-XENIX safely. If you are logged in as root, you may type:

haltsys <ENTER>

and you will see the message:

****NORMAL SYSTEM SHUTDOWN****

However, if you are in a multi-user environment, you should warn users before shutting down the system, so they can close their files and log off the system. (This is **very important** for application users.) An easy way to do this is to use the **shutdown** program to stop TRS-XENIX. At the root prompt, type:

shutdown <ENTER>

You are asked how many minutes there are until the shutdown. TRS-XENIX will send a warning message to all terminals, and proceed with its own shutdown.

HINT Defaults are set for seek rate, door open detect, and motor speed wait. These settings work on all drives, but may not be the most effecient for your drives. See Appendix C for more information.

Also, the verify mode for floppy drives (verify all writes by reading the data back and comparing) is on. This improves data integrity and minimizes the chance of lost data due to I/O problems. However, if you wish to change this setting see Appendix C.

Note: TRS-XENIX applications user's, see Appendix D for instructions on installing your applications.

CHAPTER 4

Getting the System Ready for Your Users

One of TRS-XENIX's special features is its capacity to support many users. Even if your system has only one terminal, several people may be using the system at different times. With TRS-XENIX, each user accesses, or "logs in", to the system under a separate name and is assigned a workspace, or "user directory" of their own. Each user is assigned a password to prevent unauthorized users from accessing information on the system. Your job as system manager is to create and maintain user login information.

In this chapter you will be introduced to one special user on the system: the "super-user", and learn the following simple procedures:

- how to add users to the system with the **mkuser** program
- how to remove users from the system with the **rmuser** program
- how to change a user's password

You will also be introduced to some important TRS-XENIX files: /etc/passwd and .profile.

4.1 The Super-User -- System Maintenance Mode

TRS-XENIX restricts access to many system files, and provides a protection mechanism to allow users to restrict use of their own files. There is one user, however, who has unlimited access to the system: the aptly named "super-user". The super-user logs in as "root", and is the only one who can add or remove users, for example. However, because the super-user's access is unlimited, simple mistakes can cause massive damage to system and user files. This is why it is important to assign a password to the root.

4.1.1 Assigning a Root Password

Choose a password that is not easy to guess and that is longer that six characters. Mix upper- and lower-case characters and numbers. To change the root password, type from system maintence mode:

```
passwd root <ENTER>
TRS-XENIX will respond:  Changing password
                         New password:
                         Retype new password:
```

The password will not be displayed while you are typing. After you have entered the password twice, the **passwd** command automatically updates the entry in the /etc/passwd file.

> **Important: Do not forget the root password. There is no way to recover from a lost root password except for reinitializing.**

Remember that the number of individuals who are given the super-user password should be extremely limited. Super-users should log in as root only when absolutely necessary.

Also remember that even if you are the only user on your system, you should create a login entry for yourself and log in as an ordinary user. Misuse of your super-user powers can result in disaster.

4.2 Adding a User: The mkuser Program

The **mkuser** program makes the process of adding a new user as easy as possible for you. To prevent unauthorized users from creating "logins" for themselves, you must be logged in as super-user (root) to use the **mkuser** program. At the root prompt (#), simply type:

```
mkuser  <ENTER>
```

The system will respond with the following lines:

```
Mkuser
------
Add a user to the system
Do you require detailed instructions? (y/n/q):
```

You must respond with one of these three letters:

<y> provides instructions for **mkuser**
<n> does not display instructions
<q> means "quit", the **mkuser** program terminates, and returns you to the system.

Note: Pressing the <BREAK> key terminates the **mkuser** program at any time, unless you are specifically instructed NOT to.

You will need to know the meaning of three terms in order to continue:

login name	the name by which the user is known to the system. It is generally short for easy entry. Such as a first initial and last name (jdoe), or three initials (jad). Do not exceed eight (8) characters.
comment	a space to record information about a user, the suggested format is: name, department, phone extension. This information (and format) is used in more advanced features of TRS-XENIX. You are limited to 20 characters, including spaces.
initial password	the password you, as system manager, assign to a user. The user may change this password after he has logged in with the initial password.

After you have read the instructions, you may continue by pressing <ENTER>.

You are asked to enter the new user's login name. For example,

```
login name: johnd<ENTER>
```

The program asks you for an optional password. You are encouraged to mix upper and lower case letters, digits, and special characters to make the password hard to guess. You must remember this password so that the user can log in.

Please type carefully because the password does NOT appear on your screen as you type; you cannot see if you have made a mistake. You are asked twice for the password; if the two entries are not identical, TRS-XENIX won't accept them.

After you have entered the password, you are asked for a comment entry. This entry is limited to 20 characters, including spaces.

```
Please enter Comment>--------------------
                    >John Doe,Acct,#333  <ENTER>
```

If you do not wish to enter a comment, just press <ENTER> instead. Now mkuser displays the entire entry it has created for the new user in a special system file called /etc/passwd. This entry includes the following:

user id — The mkuser program arbitrarily creates user id ("uid") numbers for each new user account, starting at 200. Numbers below 200 are reserved for system ids like root and cron.

group id — The group id is an optional facility for allowing joint access to the same files. Mkuser initially sets the group id of all new users to 50. In effect, all users belong to the same "group", and share access to all files. If greater privacy is desired, or if you wish to create true group ids, the /etc/passwd file may be edited, using your text editor.

home directory — The **mkuser** creates a work space for the new user, a "directory" of the same name as the login name, located in the /usr parent directory. In this example, the directory should be: /usr/johnd.

comment field — This will contain whatever information you have entered (in this case, John Doe's department and phone extension).

shell — The shell is a TRS-XENIX program which prompts for and interprets user commands, normally /bin/sh unless otherwise specified.

In this case, the entire /etc/passwd entry will look like this:

```
johnd:j9djfLzK8hfdo:2Ø4:5Ø:John Doe,Acct,#333:/usr/johnd:/bin/sh
```

Every user of the system must have an entry in this file. Note that every field is separated by a colon (:). The strange sequence of characters after the login name is an "encrypted", or coded, version of the password you created for John Doe.

The **mkuser** program will give you an opportunity to change the username, password, or comment at this point. If you press <BREAK>, the **mkuser** program will terminate without adding the new user. If you do not press <BREAK>, it will report:

```
Password file updated
Home directory /usr/johnd created
/usr/johnd/.profile created
User johnd added to this system.
```

Now John Doe can log on to the system using the password you assigned. If he chooses, he may change his password at this point by entering the command

```
passwd  <ENTER>
```

TRS-XENIX will respond:

```
Changing password for johnd.
Old password:
New password:
```

John enters his old password, followed by his choice of a new password. The passwd command automatically updates the entry in the /etc/passwd file.

After you have been acting in the role of system manager for a while, you may have reason to change entries in the /etc/passwd file directly, such as creating new group ids. If you are logged in as super-user, you will be able to edit /etc/passwd like any other file, with the following precautions:

- Use **passwd** to change passwords.
- Make changes to /etc/passwd when the system is relatively idle.
- Never touch system ids, like "cron" or "root".
- Save a copy of the old /etc/passwd file under another name, e.g.:

  # cp /etc/passwd /etc/passwd.back <ENTER>

4.3 The .profile File

When any user, including root, or the super-user, first logs in, a file called .profile in his "home" directory is executed automatically. This file:

- tells the system what kind of terminal the user is working on.
- establishes the name of the user's home directory.
- tells the system where to look for programs the user runs.
- tells the system where the user's electronic mail box is located.
- determines the meaning of certain characters on the terminal, such as "kill" and "backspace".

The mkuser program automatically creates the .profile file in each user's directory, with entries which should be suitable. If changes are necessary it can be edited like any TRS-XENIX file, using your text editor. Ordinary users may edit their own .profile file.

4.4 Removing a User: The rmuser Program

It is sometimes necessary to remove a user from the system. If John Doe gets a job with another company, the system manager will want to prevent him from continuing to access the payroll files. The process of removing a user includes:

- deleting the user's entry in the /etc/passwd file
- removing the user's home directory

You cannot remove a user without first removing all of his files and directories from the system (or moving them to the directories of other users). It is not a good idea to remove a user's files from the system until they have been "saved" or copied to floppy disks by one of the methods described in Chapter Eight, "Backups".

To remove all of a users files, type:

```
cd /usr/johnd  <ENTER>
rm -fr *  <ENTER>
```

and all of John's files will be deleted.

After this, you can use the **rmuser** program to remove his login from the system. **Rmuser** asks for the names of users to remove from the system. It prompts for a user name, and deletes that user's entry in the password file, removes his mail box, .profile file, and home directory.

The **rmuser** program will refuse to remove a user id or any of its files if one or more of the following checks fails:

- The user name given is one of the "system" user names, such as root, sys, sysinfo, cron, or uucp.
- The user id is below 200.
- The user's mail box is not empty.
- The user's home directory contains files other than .profile.

Rmuser can only be executed by the super-user.

CHAPTER 5

The TRS-XENIX File Structure

In order to do your work as system manager, you will need to understand how TRS-XENIX organizes the information of the system with files and directories. In the last chapter you learned about .profile and /etc/passwd and the "home" directory belonging to each user.

In this chapter the organization of files and directories on TRS-XENIX is discussed in detail. In particular, the following concepts are introduced:

- organization of files and directories
- definition of users, groups, and others
- assignment of permission to access files
- creation, mounting and unmounting of file systems

You will learn the following procedures:

- how to set and change permissions to files and directories
- how to create and mount a file system
- how to use some basic TRS-XENIX commands associated with creating, moving, and deleting files and directories and listing their contents

Finally, a road map to various TRS-XENIX files and directories of interest is presented.

5.1 Files and Directories

TRS-XENIX stores information on the disk in "files", in much the same way as you might store an important memo or record in a file folder. Computer files may contain various types of information such as the text of a document, a program, or lines of data, but they are all treated in the same way.

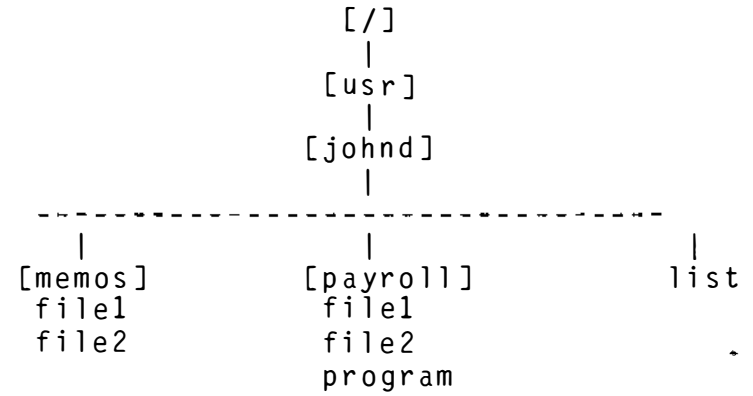
Furthermore, TRS-XENIX gives you the option of collecting groups of files in "directories", much as you might organize file folders into categories.

The organization of TRS-XENIX files and directories is quite flexible. You will soon find that your decision about what to name files and where you decide to put them can make your work on the system more efficient.

It is important to emphasize here that TRS-XENIX files and directories can be organized hierarchically. That is, each TRS-XENIX directory can contain other directories, as well as files, and those directories can also contain directories. Therefore, you can create a pyramid of directories within directories.

To take a simple example, in the preceding chapter John Doe was added to the system with the **mkuser** program. As you remember, one of the things that the **mkuser** program did was to give John a work space of his own, a "home" directory, called /usr/johnd. Now you can see that the directory johnd is, in fact, a subdirectory of another directory called usr, which contains the home directories of all the users on the system. Undoubtedly, as soon as John logs in to the system for the first time he will begin to create new files and directories to store his work in.

Let's say he creates a directory called payroll in which to put various payroll programs and data files, another called memos in which to store memorandums he receives from other users on the system and a simple file called list, which contains a list of things to do. The results would look like this (directories are indicated in brackets):



5.2 File Manipulation Commands

At this point, you will need to learn a few TRS-XENIX commands for creating, moving, copying and deleting files and directories. To create the memos directory, for example, John Doe had to use the **mkdir** command ("make directory").

```
mkdir memos  <ENTER>
```

Now, in order to make the memos directory his "current" or working directory, he would type:

```
cd /usr/johnd/memos  <ENTER>
```

The command **cd** stands for "change directory". He is now in the directory memos or, to give its full pathname, or location, /usr/johnd/memos.

To create a new file, type the character ">" followed by the file name. Let's say John wants to create a new file in his memos directory called newproject:

```
>newproject  <ENTER>
```

There is now an empty file in the memos directory called newproject.

The **lc** command lists the contents of the current directory:

```
$lc  <ENTER>
file1     file2     newproject
```

Files can be moved and copied from one directory to another. Suppose John wishes to move the file newproject to a new directory created for another project, acct.project. First, he can create acct.proj with the **mkdir** command, and then he can move the newproject file from the memos directory to the acct.proj directory with the **mv** command, as in:

```
mv newproject  /usr/johnd/acct.proj  <ENTER>
```

If he uses: cd /usr/johnd/acct.proj <ENTER>

to move himself into the acct.proj directory, and uses the **lc** command, the newproject file will appear in the list of files in the acct.proj directory.

When TRS-XENIX moves a file, it removes it from the old directory and puts it in the new one. If John wished to keep a copy of newproject in the old memos directory, thus making newproject a file in both directories, he could use the **cp** command ("copy") to create another identical newproject file in the other directory. Note that TRS-XENIX allows you to have two files of the same name, as long as they are in different directories.

If you no longer need a file, just use the **rm** command to remove it. The command:

```
rm newproject  <ENTER>
```

removes the file newproject from the directory. Once again, this can be verified with the lc command. Similarly, you can remove an entire directory by using the command **rmdir** as in:

```
rmdir memos  <ENTER>
```

However, TRS-XENIX will not allow you to remove a directory without first removing all of the files in it. You can type either:

```
rm file1 file2 newproject  <ENTER>
```

or, using a special TRS-XENIX character, the asterisk (*), which stands for everything, enter:

```
rm *  <ENTER>
```

Be cautious about using the asterisk; it really will remove everything. Also, you cannot be in the directory you are removing, so move back up to the directory above it. TRS-XENIX even lets you type two commands together, if you separate them with a semi-colon (;):

```
cd /usr/johnd;rmdir memos  <ENTER>
```

As you work more with the TRS-XENIX system, you will want to know a great deal more about these and other commands to manipulate files and directories. Read whatever other TRS-XENIX documentation you have available for a more detailed discussion of the TRS-XENIX file structure, and practice using these commands until you are comfortable with them.

5.3 Setting Permissions: Users, Groups, and Others

As system manager you will be responsible for adding and removing users on the system. Each user is given a unique password to ensure privacy, and optionally, a group affiliation that allows a number of users to share access to files. You will also need to know something about how permission to access files on the system is assigned to "users", "groups", and others on the system.

5.3.1 Users

Any user who has a valid login name and password recognized by the system may log in, and proceed to access files, edit text, or run programs. But not every user should have access to every file on the system.

For example, John Doe would probably not want every user to be able to read or change the payroll files he created in the previous section. TRS-XENIX provides a method of strictly controlling access to each individual file and directory with permission or protection settings.

Naturally, every user has access to all the files and directories which are in his home directory. When the user logs in, this is where he is initially positioned. Ordinarily, a user cannot access other files and directories unless the necessary permissions have been set.

Only the super-user has unlimited power to access files on the system, which makes logging in as "root" temptingly convenient. But since the TRS-XENIX file protection mechanism does not apply to the super-user, a single mistake made by the super-user can cause massive damage to other users' programs and data, and possibly even bring down the entire system.

5.3.2 Groups

Generally, a user will restrict access to the files and directories he owns by setting the permission for the file. This can prevent other users from changing, or perhaps even reading, his files. However, some directories and files need to be shared by several users, while still remaining restricted from everyone else. Thus TRS-XENIX offers a group id which allows more than one user the same access. In practice, you may decide not to use this feature on a small system. TRS-XENIX will automatically set the group id as if every user belonged to the same group; you may change this as desired.

5.3.3 Permissions

In order to get a look at the permission settings for all the files in a directory, we can use a variant of the **lc** command, **lc -l**. Here is a sample output:

```
-rw-rw---- 1 johnd 11515  Nov 17  14:21 file1
-rw-rw---- 1 johnd 12337  Nov 16  1Ø:15 file2
-rwxrwx--x 1 freds  7712  Oct 1Ø  Ø9:Ø2 program
```

The **lc -l** command gives you quite a lot of information. Just to the right of the name of the user who created the file, you will see the size of the file (information which will be of special interest to you when you are short on disk space and are trying to decide what files to remove), the date and time when that file was either created or last changed, and the name of the file or directory.

Now look carefully at the pattern of ten letters and dashes on the left. If the first space is a "d", it indicates a directory, a "-" means an ordinary file.

The other nine characters are a representation of the nine possible combinations of permission which can be assigned to any file. These nine can be divided into three groups of three, read (r), write (w), and execute (x) permissions for each of the following categories (left to right): the "user" who is the owner of the file, the "group" whose members have been defined by the system manager, and "other" including everyone else. In each case a dash (-) denies permission, and the letters "r", "w", and "x" allow it. The entry

```
-rwxrwxrwx  1  johnd  32 Oct 19 1Ø:ØØ  public
```

indicates that everyone, owner, group, and other, has full access rights to a file called public created by John Doe.

Of course there are some files which John might want to keep private. For example, the list file in his home directory, which contains his personal "to do" list. So, the permission setting on this file would look like this:

```
-rw------- 1 johnd     3 Nov 18 Ø7:31 list
```

Permissions are set for directories in the same way as files, but you should note these differences. The "x" permission is taken to mean "search", rather than execute in the case of directories. Search (x) permission is required to **cd** to a directory.

Search and read (r) permission are required on a directory to use the **lc** command. Write (w) permission is required to use **mkdir** or **rmdir**, or create a file in a directory.

Thus, if a directory's permission is set "r-x", you can access all of its files, subject to any permissions that might be set for any individual files. If a directory is "--x", you can only access a file if you know its name. If the directory is "rwx", you can do anything in it.

5.3.4 How to Change Permission Settings

When you add a file or directory, your TRS-XENIX system will automatically set some default permissions for you. This setting is read (r) and write (w) permission for the user, read (r) only for group and other permission setting:

```
-rw-r--r--
```

If you decide to change a setting you will need to use the **chmod** or "change mode" command. To enter the command, you will have to say whose permission is to change ("u", "g", or "o" for user, group and other), and you will also have to indicate whether you are adding (+) or removing (-) permission. After this, give the type of permission (r, w, or x), then the file or directory name. So, for example, to change the file "program" from a permission setting of read and write for user, group, and other (-rw-rw-rw-) to also allow everyone to execute the file you would need to type:

```
chmod  ugo+x program  <ENTER>
```

Or, instead of "ugo" you could type the letter "a" for all:

```
chmod a+x program  <ENTER>
```

Now, to take read and write permission away from everyone else besides you and the members of your group, you can use:

```
chmod o-rw program  <ENTER>
```

If you get the order, or syntax, of this command wrong, TRS-XENIX will sometimes answer with a bit of help:

```
Usage:  chmod [ugoa]  [+-=]  [rwxtugo]  file
```

The items in brackets are "options". You will not need the other options besides "rwx" in the last set of brackets. Anyway, always use the **lc -l** command after using **chmod** to check the results. If you make a mistake, just change the permissions again.

5.4 TRS-XENIX Files of Interest

As system manager, you will be encountering certain TRS-XENIX directories often. For now, it is sufficient to know their names and general contents. However, when you are more experienced, and decide that you want to change or delete system files or directories, it is extremely important that you check Appendix A, "TRS-XENIX Files and Directories" for more detailed information and warnings. Here is a brief list:

/bin	TRS-XENIX commands
/dev	files concerning special devices (e.g., printers, terminals)
/etc	system maintenance programs and data files
/lib	library routines used in C programming
/tmp	Temporary files created by programs, editing, and mail
/usr	user home directories and some commands

You should also become familiar with the following files and directories in the /etc and /usr directories, if they exist on your system.

/etc/passwd	containing the login name, password, group and user id of every user on the system.
/etc/motd	message of the day file, containing information the system manager needs to give all users. Message may be changed by using your text editor.
/etc/termcap	a list of terminals and their characteristics needed to support some system functions.
/etc/rc	a shell script designed to set some system functions going when the system is booted.

Chapter 6

File Systems

TRS-XENIX performs the task of disk storage management for all users of the system. It allocates disk space upon demand, keeps track of where on the disk the data is written and retrieves any part of it when given the "pathname" of the file.

When a file is no longer needed, it can be deleted and TRS-XENIX returns the space it occupied to the pool of available free space.

A disk device contains not only the files themselves, but also the information needed to locate the files on the disk and manage them. Together, these are called the "file system".

A simple TRS-XENIX system contains one disk device which is set up as a single file system, with the exception of a small area reserved for swapping, the process of switching data back and forth from the disk to the main memory of the computer.

6.1 Creating File Systems

You may find it convenient to set the system up so that a file system is resident on a floppy disk or a secondary hard disk. To do this, you must create the file system on the disk with the **mkfs** command. The disk must be formatted and write-enabled. If a file system already exists on the disk, it will be destroyed by this procedure.

Remember, each of the devices on your system has a name, and each device has a corresponding file by the same name in the /dev directory. The floppy disk drives are called "fdØ," "fd1," and so forth. (Hard disk drives are named "hdØ," "hd1," etc.)

6.1.1 Readying The Disk

Use the **diskutil** program to format your disk. Follow these instructions to format a floppy or hard disk:

```
TRS-XENIX Boot
: diskutil  <ENTER>

Copy or format (c or f) ?
     Enter f to format.

Hard or floppy disk (h or f) ?
     Enter f for floppy disk or h for hard disk

Floppy drive number (Ø..3) ?    or
Hard drive number (Ø..3) ?
     Enter the appropriate drive number (Ø-3)

TRS-XENIX format, IBM single-density
   or IBM double-density format (x or I)?
          Enter x for TRS-XENIX format.
```

Diskutil tells you to ready the disk, and to press <ENTER> to proceed (or <BREAK> to abort). Press <ENTER> and the formatting will begin. The cylinder and side numbers are displayed while the formatting is in progress. If the disk is defective, you will see the message:

```
**Format verify failed**
```

The location of the bad spot on the floppy disk will be given, followed by the message:

```
Disk is unusable.
```

IBM™ is a trademark of International Business Machines, Inc.

If you attempt to format a floppy which already has data on it, you will see the message:

```
**Destination disk is not blank**
Any data on it will now be lost if you proceed.
Type <ENTER> to proceed or <BREAK> to abort: <ENTER>
```

When the process is finished, a message is displayed saying that the format is complete. Press <BREAK> and then RESET to exit from **diskutil** and boot TRS-XENIX.

6.1.2 Creating The File System

You are now ready to create a file system on your disk. Make sure the formatted disk is write-enabled, and enter one of the following command lines. Be sure to give TRS-XENIX the correct drive name and note what type disk you have.

For double-sided floppy disk, enter the command line:

```
/etc/mkfs /dev/rfdx 2448 2 16
```

For single-sided floppy disk, use:

```
/etc/mkfs /dev/rfdx 1224 2 16
```

For eight meg hard disk, enter:

```
/etc/mkfs /dev/rhdx 16966 1 17
```

For twelve meg hard disk, use:

```
/etc/mkfs /dev/rhdx 23,Ø18 1 17
```

The x should be the drive number of the disk where you want the file system created (Ø-3). The numbers "2448", "1224", "16966" and "23Ø18" represent the total number of disk blocks on the disk. See Appendix C for more information.

> **HINT:** A disk "block" is equal to 512 bytes.

6.2 Mounted File Systems

Once you have created file systems on floppy disks, you will have to use the **mount** command to be able to access them. TRS-XENIX must be told about any other file system besides the root system, which is always present on the system.

To mount a file system on the floppy in drive Ø onto the directory called /acct.records type:

```
/etc/mount  /dev/fdØ  /acct.records  <ENTER>
```

Once you have typed this, acct.records becomes the top level directory on the floppy disk. Any new directories you create in acct.records are resident on the floppy disk. The mounted file system is attached to the root system.

Note: The directory you are mounting on (/acct.records) should be empty.

To mount a file system on hard disk 1 onto the empty directory called /h1, type:

```
/etc/mount  /dev/hd1  /h1  <ENTER>
```

Before removing the floppy file system that was mounted earlier, type:

```
/etc/umount  dev/fdØ  <ENTER>
```

This "unmounts" the file system and leaves it in a consistent state. Be sure that your current working directory is not on the floppy.

You must also "unmount" hard disk file systems (except the root). For example to unmount the above file system, type:

```
/etc/umount  dev/hd1  <ENTER>
```

> **Note:** Do NOT remove a mounted file system without first, sucessfully, unmounting it. You are likely to lose data or files if you do this.

If you have several often used file systems, you can place the mount command and the names of the file systems in the /etc/rc file, which is read by the system when it is first booted. Use **ed** to add this to the file (See Appendix W). The **shutdown** command automatically unmounts all devices before shutting down the TRS-XENIX system.

CHAPTER 7

System Maintenance

Keeping your TRS-XENIX system running smoothly depends largely on two factors:

- maintaining file system integrity, that is, having "clean" file systems
- ensuring that adequate free disk space is available to the users

This chapter introduces you to a number of TRS-XENIX tools which aid you in performing these maintenance tasks. These programs help you clean up files and file systems by reporting disk space usage and locating little used files. Also discussed are some system checking programs that TRS-XENIX runs automatically, and the various ways you can communicate with the other users on the system.

7.1 Processes

All functions running on the system -- including system and user programs, editing, etc. -- are "processes." Several users may each have several processes running simultaneously and it is often necessary to check whether certain processes are running, in order to stop or "kill" them. The **ps** or process status, command can be used to list the processes currently running. The output of **ps** is the list of processes running from the terminal at which you type the command. However, if you specify the parameter "-a":

```
ps -a
```

your output is "all" the processes running on the system:

```
PID   TTY   TIME CMD
 2    co    1:Ø7 -sh
73    Ø1    2:Ø9 tsh
74    Ø1    Ø:1Ø sh -c ps-a
77    co    Ø:59 ps -a
```

The first column is the process id, the name by which TRS-XENIX identifies the process. The second column is the

terminal number where the process is running (co=console, Ø1=terminal 1, etc); note that several processes may be running from a single terminal. The next column is the total time the process has run. Finally, the name of the command or program is given. You may often have use for the **ps** command while doing system maintenance.

7.1.1 Daemon Processes

In addition to those programs which you will use in the course of your system maintenance work, you should be aware that there are also a few other programs that run automatically whenever you use your TRS-XENIX system. These are called "daemons" (pronounced "demons"). These daemon programs periodically check the system or perform basic systems functions. Some examples of daemons are:

- **update** "updates" the disk by automatically writing information from memory back to disk every thirty seconds. This ensures that in the unlikely event that your system "crashes" or halts abnormally, the information recorded on disk is as current as possible.
- **lpd** superintends the operation of the line printer.
- **cron** acts like an alarm clock, allowing you to execute commands and jobs at times you specify in advance. It repeatedly looks in a file called usr/lib/crontab for instructions to perform these functions.

7.1.2 /etc/rc

Ordinarily, the commands to start these daemons are put in the /etc/rc file, which runs automatically at the time you boot TRS-XENIX. The /etc/rc file also contains directions for the system to perform other functions. For example, it may contain a message that greets you when you log in, or direct the system to ask you for the time. **Mount** commands for often used file systems may also be placed in /etc/rc, so that these file systems are automatically mounted when the system is booted. You must be logged in as root to edit the /etc/rc file.

7.2 The Importance of Disk Space

Available disk space rapidly becomes a precious commodity on any computer system. As users compile programs, edit files, or perform other tasks, they are competing for this valuable resource. On a typical system, the potential for running out of free disk space is very high, and when this actually occurs, no new files can be created, nor can any existing files expand.

If possible, each file system should contain approximately 15% free space, more if usage of the file system fluctuates, less if it is relatively stable. It is extremely important that you anticipate the risk of running out of space. Regard the task of monitoring disk space as an essential part of preventive maintenance.

TRS-XENIX offers some techniques for finding out how much free space exists in a particular file system, and determining which files might be candidates for deletion of there is a shortage of space. The TRS-XENIX commands that will help you determine the status of disk space on a file system and help you remove unwanted files are:

df	disk free
du	disk usage

Brief descriptions of these are given next.

7.2.1 The df Command

This command prints out the number of free blocks available in whatever file system is specified. If no file system is specified, the free space in all normally mounted file systems is printed. You can just type:

```
df  <ENTER>
```

or you can specify a file system:

```
df /dev/root  <ENTER>
```

The output will look something like:

```
/dev/root 7ØØ8
```

This indicates that the root file system has 7ØØ8 free disk blocks. Naturally, you must know the size of your hard or floppy disk in blocks to determine what percentage of the total disk these numbers represent (see Appendix C). Remember, a block is equal to 512 bytes.

With experience, you will come to know your system usage well enough to know whether disk space is critical.

7.2.2 The du Command

Let's suppose that you discover when you use the **df** command that you have a shortage of disk space. Now you have to do some detective work to find out where space may be wasted on your system.

The command **du** gives the number of blocks that are used by files in the specified directory and each of its subdirectories. If you use **du** without specifying a file name, it reports the size in blocks, of every directory and file, starting at the current directory. Or, you can give a file or directory name. In this case, you would probably search the /usr file system for excessively large files. Part of the output might look like this:

```
 du /usr  <ENTER>
2Ø8       /usr/anthonys/admin
378       /usr/anthonys/mp
999       /usr/anthonys/junk
1585      /usr/anthonys
26        /usr/johnd/memos
235       /usr/johnd/payroll
261       /usr/johnd
```

7.2.3 The quot Command

The quot command is another useful reporting tool. It prints the number of blocks currently owned by each user in the named file system. Typing:

```
quot -n /dev/hdØ  <ENTER>
```

will give you a list of all files and their owners in the /dev/hdØ file system.

7.3 Communicating with Other Users

Enlisting the cooperation of your users is, in a way, one of your most powerful maintenance tools. Communication with the other system users can aid you in your search for free disk space.

For example, the /etc/motd, file, which contains the "message of the day," can be edited to gently remind users that space is low and that old files should be deleted. You should edit the /etc/motd file daily, so that the users don't come to expect old and unreliable information.

You can also use the **wall** (write all) command to tell all users who are presently logged in:

```
wall
There is a shortage of free disk space.
Clean  up your unused files.  <CTRL><D>
```

Try to reserve the use of **wall** for emergencies, however, because it disrupts other users.

7.4 File System Integrity

In addition to the problem of maintaining adequate disk space, there is the possibility that a file system may develop inconsistencies. A file system consists of files which consist of blocks of bytes. If a block of information is bad, then the file, and potentially the entire file system, is compromised.

A file system's integrity is compromised when it is internally inconsistent. This occurs either when the system has not been shut down properly or if there is a hardware error due to faulty disk drives or floppy disks.

A program called **fsck** or file system check, is the tool TRS-XENIX uses to check the consistency of file systems and, if necessary, repair them. If the system was not shut down properly, the next time you attempt to boot the system you are asked if the system should proceed with "cleaning." You should always answer <y>. If your file system is "dirty," that is, inconsistent, all your files are at risk.

The **fsck** program is responsible for file system cleaning. If you have reason to suspect that file system inconsistency is responsible for any abnormal behavior, you should invoke **fsck**, followed by the name of the questionable file system. However, you should use **fsck** cautiously since it occasionally deletes damaged files during the cleanup.

Here is an example:

```
fsck -y /dev/hdØ  <ENTER>
** phase 1 - Check Blocks
** phase 2 - Check pathnames
** phase 3 - Check connectivity
** phase 4 - Check Reference Counts
** phase 5 - Check Free List
 426 files 6753 blocks 7ØØ8 free
```

CHAPTER 8

Backups

The importance of "backing up," or copying, the contents of your system regularly cannot be overemphasized. Backups are the only insurance your users have against time-consuming and costly losses of their programs and data. You should no more consider skipping a scheduled backup than you would consider skipping an insurance payment.

In addition to inevitable, every day user mistakes -- deleting, changing, or writing over files, there are more catastrophic possibilities: accidental damage to TRS-XENIX system software or hardware failure.

If you are lucky, you will rarely need to use your backup copies, but nonetheless you must have a systematic plan for scheduling backups. You must also determine how often a full backup is required, and deciding where, and for how long, you should store your backups. Some suggestions are provided here, but you will have to assess the needs of the users on your own system.

The TRS-XENIX system offers several alternative ways of making copies of system and user files. Which method or methods you choose will depend on how often the contents of your disks change significantly and the total size of your system. The following procedures are discussed in this chapter.

- formatting and copying disks with the standalone diskutil program
- using the **tar** program to copy the contents of a file system
- using the sysadmin program to simplify backup and restore functions

8.1 Formatting Floppy Disks

You must format and copy floppy disks with the same standalone diskutil program which you used to install TRS-XENIX on the hard disk. Therefore, it is necessary to bring your TRS-XENIX system down by logging in as root and typing

shutdown <ENTER>

Of course, you will want to minimize the need for shutting down the system, especially if you have several users. The system manager should plan to format spare floppies in advance and perhaps copy disks for other users when the system is not in use.

> Note: Every floppy disk you format or copy must have a write-enable tab.

You will be using the diskutil program in much the same way you did when you first installed TRS-XENIX on your hard disk. The diskutil will prompt you during either of the following procedures:

format hard (8 or 12 megabyte) or floppy (single- or double-sided) disks

copy a single or double sided TRS-XENIX floppy disk

Here is what you would see on your screen during a sample diskutil run to format floppy disks:

```
TRS-XENIX Boot
: diskutil  <ENTER>

Copy or format (c or f) ? f <ENTER>   (format)
Hard or floppy disk (h or f) ? f <ENTER>  (floppy)
Floppy drive number (Ø..3) ? Ø <ENTER>  (Drive Ø)
```

```
TRS-XENIX format, IBM single-density
   or IBM double-density format (x or I)? x <ENTER>
                                          (trs-xenix)

About to format floppy disk in TRS-XENIX format.
Insert disk in drive Ø.
type  <ENTER> to proceed or <BREAK> to abort:<ENTER>

This will take about 77 seconds.
```

Press <ENTER> and the formatting will begin. The cylinder and side numbers are displayed while the formatting is in progress. If the disk is defective, you will see the message:

```
**Format verify failed**
```

The location of the bad spot on the floppy disk will be given, followed by the message:

```
Disk is unusable.
```

If you attempt to format a floppy which already has data on it, you will see the message:

```
**Destination disk is not blank**
Any data on it will now be lost if you proceed.
Type <ENTER> to proceed or <BREAK> to abort: <ENTER>

Floppy disk in drive Ø successfully formatted.
```

At this point you will be given the opportunity to repeat the formatting process on another floppy. Type <BREAK> if you have no more floppies to format, then press RESET to exit from **diskutil**.

IBM™ is a trademark of International Business Machines, Inc.

8.2 Copying Floppy Disks

The procedure for copying data from one floppy disk to another with the **diskutil** program is quite similar. **Diskutil** will automatically format disks before it copies data to them. After you enter diskutil, simply type <c> for copy instead of <f> for format. You will be asked for the number of the "source" and "destination" drives:

```
Source drive number (Ø..3)?
Destination drive number (Ø..)?
```

The "source" is the drive that contains the floppy disk to be copied. The "destination" is the drive containing the disk where the informations is being copied to. Respond with the correct drive numbers.

The next message provides you with more directions.

```
Insert source disk in drive Ø.
Insert destination disk in drive 1.
Type <ENTER> to proceed or <break> to abort.
```

If you are copying to a disk which is already formatted or contains other data, you receive a warning message:

```
**Destination disk is not blank**
Any data on it now will be lost if you proceed.
```

You will see the number of cylinders and sides change as the copying takes place. When the copy is finished you will see the message:

```
Disk copy and verify complete
```

8.3 When to Make Backups: Daily and Periodic Backups

The **sysadmin** program allows you to perform two levels of backup. For simplicity, these are called "daily" and "periodic." Although a suggested schedule might include a "daily" backup every day and a "periodic" backup once a week, you should decide what is appropriate for your system. This normally depends on how heavily the system is used, and how often the majority of files are changed. A periodic backup should be done at least once a month.

It is probably a good idea to keep other users off the system while you are doing backups, if this is at all possible. At the very least, there should be little or no activity on the system -- to avoid changes to a file while it is being copied. Needless to say, backups should be scheduled so that they have the least possible impact on users.

When you do a full backup you are copying the entire root file system to floppy disks. Since a relatively small number of the files in a file system change frequently, however, it is possible to supplement your periodic backups with "daily" backups. The daily backup is incremental; that is, it only makes a copy of those files which have changed since the last periodic backup.

8.4 Archiving and Taking Care of Your Disks

You will rapidly accumulate a great number of floppy disks. These will include:

TRS-XENIX distribution disks

application packages

user file systems

backups

Each full, or "periodic," backup of the TRS-XENIX system may require a large number of floppy disks (e.g. seven or eight double-sided disks to backup a TRS-XENIX system with an eight megabyte hard disk). You must have enough disks on hand to keep the backup disks in reserve for several weeks, as well as keeping sufficient spares on hand for your users.

In addition, you should develop a simple but logical method for organizing, labeling, and storing your disks. Since your floppy disks will contain valuable, and in some cases, irreplaceable data, a high priority should be given to safety and security. You should consider designating an off-site storage area in which to keep alternate backups, to enable recovery from a disaster which damages your computer area.

You should develop a consistent policy for saving your backups; resist the temptation to immediately reuse the backup disks. Anticipate that users will occasionally ask for the restoration of relatively old files. One approach is to save the full or "periodic" backup for an indefinite period after they are made, and the incremental backups for at least two weeks. You can easily rotate these disks back into use for new backups.

8.5 Using the tar Command

The **tar** program is a convenient way to copy a small number of files or directories to floppy disks. To use **tar**, make sure you are logged in as root, insert a formatted floppy disk in a floppy drive, and type the command in the following form:

```
tar cvf /dev/rfdØ file1 file2 file3 <ENTER>  (Drive Ø)
```

or:

```
tar cvf /dev/rfd1 dir1 dir2 dir3 <ENTER>  (Drive 1)
```

Note that you need not use either **mkfs** or the **mount** command to copy to this disk. If you are copying directories, the whole directory tree starting from the directory or directories you name will be copied. To get your copies back, position yourself in the directory in which you need to copy the files or directories and type:

```
tar xvf /dev/rfdØ  <ENTER>    (Drive Ø)
```

8.6 Using the Sysadmin Program

The **sysadmin** program will work on any hard disk TRS-XENIX system. Logged in as root, type the command:

```
/etc/sysadmin  <ENTER>
```

Sysadmin is a script for performing file system backups and for restoring files from backup disks. It can do either a "daily" backup or "periodic" full backup.

It also has an option to provide a listing of the files backed up, and a facility for restoring individual files from backup disks.

Sysadmin will work with TRS-XENIX formatted disks, either single- or double-sided. By default, it backs up the root file system. The script can also be edited to backup additional file systems, if required. Optionally, you can list the names of the files which have been backed up and store them in a file called /tmp/backup.list.

8.6.1 How to Do a Backup With Sysadmin

If you are doing either a daily or periodic backup, first be sure that you are logged in as root. Be sure you have on hand a sufficient number of blank, or reusable, formatted floppy disks with write-enable tabs. In response to the prompt, type:

```
/etc/sysadmin  <ENTER>
```

The **sysadmin** program will produce a menu of "file System Maintenance" tasks for you:

```
                File System Maintenance
                -----------------------
        Type    1 to do daily backup
                2 to do a periodic backup
                3 to get a dump listing
                4 to restore a file
                5 to quit
```

Answer with a <1> or <2> depending on the type of backup you want. You are asked to indicate whether you are using single- or double-sided disks.

Then insert a formatted disk in drive Ø and press <ENTER>. The system will respond with the current date, followed by the date of the last system dump.

> **Note:** If the system has never been backed up, you will see the phase "the epoch;" in this case, everything on your TRS-XENIX system will be backed up, even if you chose the "daily backup" (number 1) from the sysadmin menu.

As the backup proceeds, a series of messages will come to your terminal. The only one that need concern you is the instruction to "change volumes." When you see this message, remove the disk presently in the drive and replace with another. You should number your disks during this procedure (backup disk #1, backup disk #2, etc).

Press <ENTER> after new disk has been inserted. You will probably need to repeat this process several times before the backup is complete. Do NOT remove any disk until you see the "change volumes" instruction.

If you make a mistake at any time in the process, you will have to begin the backup again. You may receive a message like:

```
dump:  write failure on /dev/rfdØ
```

This occurs if you are using a disk which has not been correctly formatted, or if your floppy disk drive has malfunctioned during the backup process. You are returned to the super-user system prompt. Type /etc/sysadmin again, to re-enter the **sysadmin** program.

8.6.2 Getting a Backup Listing

You may find it useful to keep a record of the files you have backed up. To do this, return to the **sysadmin** program after the daily or periodic backup and type <3> to receive a backup listing. You are prompted to re-insert the backup disks in the same order that they were during the backup. Sysadmin will read the file names off the backup disks and place them in the file /tmp/backup.list. You should be able to print this list on your printer.

8.6.3 Restoring a File

If you need to restore a file from a backup, enter **sysadmin** and select option <4>. Unless you have experienced a widespread disaster, you should restore files one at a time. When you restore a TRS-XENIX file, it is assigned a number rather than its original name. This number is unique to a single TRS-XENIX file.

The **sysadmin** program cannot recognize the difference between files of the same name in different directories because it does not know about pathnames. The file you restore is placed in your current working directory, and so to avoid possible confusion, rename it to its original name and move it back to its correct position in the directory hierarchy (using the **mv** command).

CHAPTER 9

TRS-XENIX System Security

Although security is more of an issue on larger systems, every system manager must take into consideration the protection of data and programs from unauthorized access. This concern may be present whether you have two or ten regular users, but even if you have only one terminal attached to your system, you may have different users authorized to access the system for different tasks at different times.

The first step is to use TRS-XENIX tools to maximum advantage. Intelligent decisions about the following can make a big difference in overall system security:

- the establishment of user accounts and group ID
- initial password selection and changing of passwords
- the assignment of permissions to maximize file protection, while still allowing convenient access to user files and directories.

In addition, you should consider taking physical security precautions:

- removing the key from the hard disk
- organizing and locking up floppies
- making sure all your users log out when they are finished working
- protecting the work area from intrusion
- protecting the computer and disks from physical damage through exposure to cigarette smoke, spilled drinks, or the use of ball point pens to make notations on disk labels, etc.

- keeping some backups off-site, in case of disaster in the computer area.

Once again, be sure that you limit the use of the super-user login, to minimize the risk of accidentally damaging system files and programs.

9.1 Protection and Permission

When considering the assignment of permissions, note the special case of access permission for directories. Remember that the execute-permission bit for directories grants permission to search the directory for a given file during the scanning of a pathname.

If a user has execute permission, but not read permission, to a given directory, he may access files in that directory, even though he is unable to read the contents of the directory. Write permission of a directory means that the user may create and delete files in that directory.

The capacity of the "super-user" to read and write to any file in any directory, and change any permission settings can present serious security problems. The root password can only be given to individuals who are authorized to read and change any file in the system. Also, the use of the root login should be severely restricted because of potential damage to the system. Even the system manager must be extremely cautious about operations undertaken while logged in as root.

Needless to say, the system manager must be careful to assign the correct protection to the files under their control. In particular, it is necessary that special device files be protected from writing, and probably reading, by ordinary users when sensitive files belonging to other users are stored on the system.

> **Note:** Programs can be written to examine and change files by accessing the device on which the files are resident.

9.2 Password Security

You should discourage users from choosing passwords that are easily remembered, and hence readily guessed: these tend to be short, from a limited alphabet, often found in the directory, and frequently something obvious like the user's own nickname or license plate number. They should be reminded that this defeats the idea of having passwords.

Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

9.3 Restating the Obvious

The most obvious but often neglected security problem results from users failing to log out of the system when they finish their work, leaving all the files to which they have access vulnerable to unauthorized use. Inexperienced users should be repeatededly reminded to log out.

Restricting access to the work area, locking up archives of floppy-based data and programs, and removing the key to the hard disk are simple protective measures that can be taken if a high level of security is required.

CHAPTER 10

Troubleshooting

If you follow the instructions in this operations guide carefully, you will have few problems with your TRS-XENIX system. However, some difficulties are inevitable with any computer system. Included here is a discussion of how to deal with some common system problems.

10.1 Jammed Line Printer

You will need to be super-user to carry out these instructions. Check to see which process "owns" the line printer with the **ps -a** command. Remove it with the **kill** command. Remove the file /usr/spool/lpd/lock and queue another print job, by typing:

```
ps -alx    (search the output for the right process).
kill  -9 [process id number]
cd /usr/spool/lpd
rm -f lock
pr  file|lpr&
```

10.2 Forgotten Password

The system manager -- or someone who has the authority to log in as super-user -- must create a new password:

```
passwd joeb  <ENTER>
New passwd:
Reenter passwd:
```

This will give a user called "Joeb" a new password.

10.3 System is Out of Space

This requires some work. When there is no free space left on the designated file system use the system maintenance tools described in Chapter Six to find and delete unused files. If you are chronically short of space, remind users regularly to clear up their files, using the message of the day file, /etc/motd.

1Ø.4 System Files Damaged

You must be in "system maintenance" (single-user) mode to do the following. That is, when you boot, instead of typing <CTRL><D>, type in the root password. Follow the normal procedure for restoring files; in this case, the mount prefix is /. So if /etc/passwd is lost or damaged, it is can be recovered by using the following:

```
restore xf /dev/rfdØ /etc/passwd
```

In this example, the **/etc/passwd** file is restored from the disk in floppy disk drive Ø.

1Ø.5 Terminal Difficulties

If you use the <BREAK> to exit from a screen-oriented program, there is a possibility of leaving the terminal in a "no echo" or "raw" mode. This is very confusing, because you cannot see any output to the terminal. Try the following to reset the terminal:

```
<CTRL><J>
stty echo -raw <CTRL><J>
```

Note that the first time you type <CTRL><J>, you may get an error message. Just try it a second time. The **stty** command is terminated with a <CTRL><J>, not an <ENTER>. If you are in "no echo" mode, of course, you will not see the command as you type it.

1Ø.6 Forgetting the Root Password.

Don't. The system will have to be re-installed.

1Ø.7 Removing a Directory

When using the **rmdir** command to remove a directory, you may get a message saying that the directory is not empty, even though the **lc** command does not list any files. Remember that there may be one or more files beginning with "." (.profile, for example) which do not show up in **lc**

output. The command **rm** * will NOT delete these; the asterisk (*) will not pick up these files. Use:

```
lc -a <ENTER>
```

to see these files and then, type:

```
rm .*  <ENTER>
```

to remove them. (Note that every directory also contains two files called "." and ".."; you can not remove these, but they do not have to be deleted in order to remove the directory.)

10.8 Special Characters in File Names

You should never use any of the following special characters in the name of a file or directory:

```
< > . / ? { } ' " ; | - ( ) * & ^ $
```

The worst case is beginning a file name with a dash (-). The **rm** command thinks the "-" is a flag and reports an error. It can be difficult to remove a file whose name contains any of these special characters. If **rm** won't work, try renaming the file with the **mv** command, as in the following case where a file is named -x:

```
mv -x junk  <ENTER>
```

Then remove junk. As a last resort, move everything else to another directory, and do

```
rm -rf dir  <ENTER>
```

on the old directory. Then use **mv** to rename the temporary directory to the old name. Do not try to delete oddly named files with wild card characters like asterisk (*) and question mark (?) unless you have thought it out carefully. You can easily end up deleting everything in your directory.

1Ø.9 Runaway Processes

You may occasionally find yourself cursed by "runaway processes" -- processes that you cannot stop, or unwanted output coming to your terminal. Try the following, in this order of preference:

1. Wait until the process finishes. This is by far the safest course of action, unless the process is causing harm.
2. Try pressing <BREAK>.
3. A process may prevent you from accessing the terminal. (e.g. it may either be running in the foreground and not accepting input, or running in the background and producing lots of output to the screen. In this case, go to another terminal (if you have one) and run

   ps -al <ENTER>

   to determine the process id (PID column) of the unwanted process. Then try the following, until the process goes away. The third command is sure to work, but may leave temporary files in the system, or leave the terminal in an unknown state.

   $ su root <ENTER>

   # kill -2 [process id number]
   # kill -3 [process id number]
   # kill -9 [process id number]

   between each kill command, wait a few seconds to see if the process terminates, before trying the next one.

4. In the event that a program prevents you from using the system, and you don't have another terminal from which to issue the kill command, you may have to resort to resetting the machine. This should only be done if ABSOLUTELY necessary.

   Be sure to wait a couple of minutes before resetting the system so the disk buffers can be flushed to the hard disk. This way the system will be as consistent as possible when shutdown in this abnormal state.

Appendix A: TRS-XENIX Files and Directories

If you consider changing, moving, or deleting any TRS-XENIX system files or directories, it is absolutely essential that you read the following information very carefully. There are, of course, several cases in which you might decide to make these modifications, but you must proceed very cautiously:

- You may wish to remove unused commands or other files from a crowded disk, to make room for your own applications.
- You may wish to edit files to adapt the system to your own requirements. An example of this might be adding an entry in the /etc/termcap file, if you are using an unusual, previously unsupported terminal. See Appendix B, "The Multi-User System."

Naturally, you should not attempt any modifications unless you have done a full backup of your system. Also, note the list of files in the following section. You should not touch these under ANY circumstances. The loss of these files is irrecoverable; you may not even be able to boot your system again. If any essential files are lost, you will need to recreate the initial system from the distribution disks (i.e. following the installation procedure described in Chapter Three, "Starting Up", then booting the system again and restoring all your files from the most recent backup.

A.1 Do Not Touch

You may have some, or all, of the following files on your system. Do not remove them.

/fdboot	/hdboot	/diskutil
/xenix	z80ctl	

A.2 /bin

The /bin directory contains all executable TRS-XENIX commands. The following should not be removed from the directory.

basename	echo	passwd	su
cp	expr	rm	sync
date	fsck	sh	tar
dump	login	sleep	restor
dumpdir	mv	stty	

One final note: do not remove a file with the peculiar name "[" from the /bin directory; it is required for the operation of system shell scripts.

A.3 /dev

This directory contains special device files which control access to peripheral devices. You should not delete or change any of them, since they are used by essential TRS-XENIX commands.

/dev/console	system console
/dev/fdØ	floppy drive Ø
/dev/fd1	floppy drive 1
/dev/hdØ	hard disk Ø
/dev/lp	line printer
/dev/mem	physical memory
/dev/null	null device (used to redirect unwanted output)
/dev/rXX	unbuffered interface to corresponding device name

/dev/root	root file structure
/dev/swap	swap area
/dev/ttyXX	terminals
/dev/tty	the terminal you are using (the system will supply the number)

Please note that you should never rename any of these files. The system relies on some of these names. However, you can use the ln command to link a name variant. For example, use the command:

```
ln /dev/fdØ /dev/floppyØ
```

If you do accidentally destroy a device special file, you will have to restore it from a backup done with the dump command. The tar command will not handle special device files.

A.4 /etc

The /etc directory contains miscellaneous system date files,as well as administrative and other system programs. Some of these are:

/etc/mtab	mounted device table
/etc/passwd	password file
/etc/mount	for mounting a file structure
/etc/mkfs	for creating a file structure
/etc/init	first process after boot
/etc/rc	bootup shell script

You should not touch any files in the /etc directory, except /etc/ttys to add or subtract terminals from your system, and /etc/termcap to add terminal types. You will also want to edit /etc/motd, the file containing the message of the day which is sent to all users when they log in.

A.5 /lib

It is not a good idea to delete anything from this directory, although it contains mostly libraries for the C compiler. If you are using the C compiler you will need them.

A.6 /mnt

This is an empty directory for mounting other file systems.

A.7 /tmp

This directory contains temporary files, many of which can be deleted if you are short of disk space. Naturally, you should check these individually before removing them while processes are still running. You can also put a command in /etc/rc to do this automatically at the time of each boot:

```
rm -fr /tmp
mkdir /tmp
chmod 777 /tmp
```

A.8 /usr

In addition to all the users' home directories, the usr directory contains the following:

/usr/bin	Contains more commands, generally those less frequently used or non-essential to TRS-XENIX system operation

/usr/include — Contains header files for compiling C programs. Can be deleted if you are not using the C compiler.

/usr/lib — Contains more libraries and data files used by various commands; should not be deleted.

/usr/spool — Contains various spoolers which store files in directories (e.g., /usr/spool/lpd, etc.).

/usr/tmp — Contains more temporary files which might be deleted.

/usr/adm/messages — Contains a record of all the console messages. Typically, these are reports of disk errors, along with some record of user errors, such as "out of disk space" messages. The messages are useful for determining whether you have a hardware problem; you can scan a week's worth of messages and see, for example, if one particular drive is generating an unusual number of errors. You will not see every disk error as it appears on the console screen.

HINT This file is likely to grow quickly, so periodically check it, print it out, and delete the file. Save the printout as a record of your problems.

Appendix B: The Multi-User System

If you have a TRS-XENIX system with a hard disk and several terminals, you will be able to use a number of TRS-XENIX commands intended for systems with several simultaneous users. These include commands which allow you to determine which other users are presently on the system, such as **who**.

You will also be able to communicate with other users on the system with **wall**, which broadcasts any message you type after the prompt.

However, there are several special cautions that apply to Multi-user system environments. These are described here.

B.1 Bringing Down the System

You should be extremely cautious about shutting down your **TRS-XENIX** system when you have several users, since you may cause damage or loss to their work in progress. Use the **who** command to find out if anyone else is logged in. Even if no one is actually logged in, processes may still be running on the system. The output of the command

```
ps -a <ENTER>
```

should be checked carefully to determine whether this is the case. Users can be alerted that you are about to bring down the system with the **wall** command. The best way to bring down the system is with **shutdown** which automatically warns users, and lets you set a timer for when the shutdown should occur. Type:

```
shutdown  <ENTER>
```

The program /etc/haltsys can also be used.

B.2 Setting Up Multiple Terminals

In order to use your system, you must tell TRS-XENIX what type of terminal you are using. By default the console is set to "trs16" terminal mode and both terminals (tty01,tty02) are set to the Adds25 emulation mode of a DT-1 terminal.

This information is stored in the file /etc/ttytype. To change the terminal types, edit the /etc/ttytype file to include one of the terminal descriptors listed in the /etc/termcap file.

The /etc/termcap file stores the characteristics of all commonly used terminals and is called by the user's .profile file at login time. The following list gives the names and codes for the terminals pre-configured in /etc/termcap file:

NAME	CODE
VT 100	vt100
VT 52	vt52
ADM 3a	adm3a
ADM 5	adm5
Televideo 910	tvi910
ADDS 25	adds25

Another default of TRS-XENIX is that the console is enabled and both terminals are disabled. To enable a terminal, type:

```
enable tty01  <ENTER>      (to enable serial channel A)
enable tty02  <ENTER>      (to enable serial channel B)
```

To disable a terminal type:

```
disable tty01 <ENTER>   or    disable tty02  <ENTER>
```

B.3 Connecting a Data Terminal to Your Computer

1. Connect one end of a DB-25 Cable (26-44Ø3) to the DT-1 RS-232 Jack.
2. Connect the other end of the above cable to the female plug of a Null Modem Adapter (26-1496).
3. Connect the male plug of the adapter to Serial Channel A or B of your computer.
4. If your computer requires it, insert a Terminator Plug into any unused Serial Channel.

More information on connecting a DT-1 to your computer can be found in the Data Terminal Owner's Manual.

B.4 Setting Input/Output Parameters for the DT-1

Follow these instructions to set the I/O parameter for the DT-1 Terminal

1. Connect the DT-1 as described earlier. You must use the null model (26-1496) as described.
2. Power up your computer system and the DT-1.
3. Set the Input/Output Parameters. Type:

   <CTRL><SHIFT><ENTER>

   to display the I/O Parameter Menu.
4. Set the parameters to the following:

   Ø Ø 1 Ø Ø Ø 1 Ø
   Ø Ø Ø Ø 1 Ø Ø 1
   Ø 1 Ø Ø 1

5. Enable the terminals as described in the TRS-XENIX Operations Guide

B.5 Setting Your System For Remote Use

The following instructions describe how to set up your system for use from a remote location. The baud rate is set for either 1200 or 300 baud.

1. Edit the etc/ttys file (with **ed**) to look like this:

```
lhconsole
03tty01
03tty01
```

   The '3' in the second column indicates that the terminals are remote.

2. Enable the necessary tty ports as described earlier.

3a. For 300 baud, set the I/O parameters to the following:

```
0 0 1 0  0 0 1 0
0 0 0 0  0 1 0 0
         0 1 0 0 1
```

3b. For 1200 baud, set the I/O parameters to the following:

```
0 0 1 0  0 0 1 0
0 0 0 0  0 1 1 0
         0 1 0 0 1
```

4. Now, when you log into TRS-XENIX, the system will automatically switch between 1200 and 300 baud until it receives something it understands. You may get some garbage displayed. If this happens, press <ENTER> until the word "login:" appears.

B.6 Setting User ID For Multiple Systems

If you are working in an environment with several TRS-XENIX systems, and users wish to work on more than one computer, moving their files back and forth on floppy disks, they will need to have valid logins and "home" directories on each system.

Please note, however, that the user must be assigned the same user ID number on each system if he wishes to transfer file systems between computers. If you customarily add user accounts with the **mkuser** program, be aware that the assignment of a user id number is done automatically and arbitrarily. Therefore, you must edit the /etc/passwd file to supply identical user id numbers for each of your systems.

Appendix C

Defaults of the TRS-XENIX System

The following information summarizes the defaults set by TRS-XENIX. Where applicable, details on how to change the default have also been included.

C.1 **The Root Password** -- The initial setting of the root password.

Default: <ENTER>
Files Affected: /etc/passwd
To Change: use **passwd** command (see Chapter 4)

C.2 **Adding Users with the mkuser command** -- default settings used by **mkuser**.

C.2.1 **Group and User Numbers**

Default: user# = sequentially assigned numbers above 200
group# = 50
Files Affected: /etc/passwd
To Change: use **chown** or **chgrp** to alter owner number or group number

C.2.2 **Standard .profile File**

Default: creates standard .profile for each user added with **mkuser**.
Files Affected: /usr/lib/mkuser.prof
To Change: edit the /usr/lib/mkuser.prof file

C.3 Drive Defaults

C.3.1 Number of Blocks and Kbytes Per Disk

Type of Disk	Blocks	Kbytes
single-sided floppy disk	1216	608 Kbytes
double-sided floppy disk	2448	1224 Kbytes
Eight Meg Hard Disk		
Primary	14,909	7454.5 Kbytes
Swap Area	2057	1028.5 Kbytes
Secondary	16,966	8483 Kbytes
Twelve Meg Hard Disk		
Primary	20,961	10480.5 Kbytes
Swap Area	2057	1028.5 Kbytes
Secondary	23,018	11,509 Kbytes

C.3.2 Mounted Drives -- tells TRS-XENIX which drive devices to mount to the root file system.

Default: Only the primary hard disk is mounted (/dev/hd0)
Files Affected: /etc/mtab
To Change: Edit /etc/rc file to include a mount statement.

C.3.3 Write Verifies -- tells TRS-XENIX whether or not to verify all writes.

Default: Verify is turned off.
Files Affected: none
To Change: Type **verify y** to turn on verify.

C.3.4 **drive Command** -- tells TRS-XENIX what type of floppy disk drives are on the system.

Default: Drive Ø: 1Øms,detect,wait
Drive 1-3: 15ms,nodetect,wait
Files Affected: /xenix
To Change: issue a drive command using the following syntax:

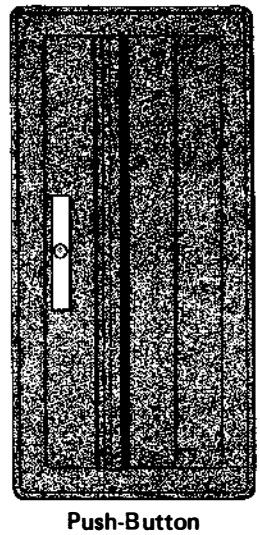
drive [#] [options]

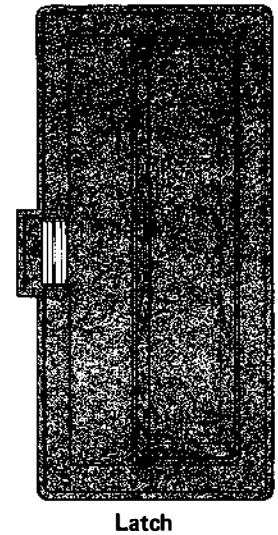
options available are:

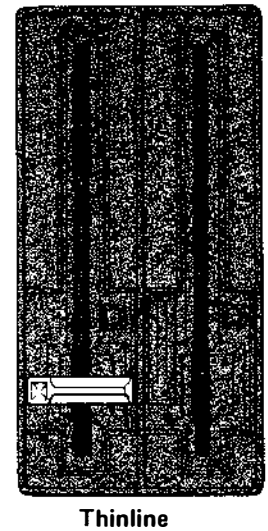
rate=n - sets the seek rate. n can be Ø,1,2 or 3 for 3ms, 6ms, 1Øms or 15ms respectively
detect|nodetect - sets the door open detection on or off.
wait|nowait - sets "wait for drive motor to reach proper speed" on or off.

Suggested settings:
Thinline = Ø,detect,wait*
Latch = 3*,nodetect*,nowait
Push-Button = 2,detect,nowait
* indicates a required setting

This procedure changes the settings for the next time you boot up.


Push-Button


Latch


Thinline

C.4 Terminals

C.4.1 Terminal Settings -- tells TRS-XENIX what type of terminal is expected.

Default: Console = "trs16" terminal mode
Both terminals (tty01,tty02) are set to Adds25 emulation mode of a DT-1 terminal.
Files Affected: /etc/ttytype
/etc/termcap
To Change: Edit the /etc/ttytype file. Terminal descriptor must be listed in the /etc/termcap file.

C.4.2 Terminal Status -- tells TRS-XENIX which terminals user can log in at.

Default: console = enabled
tty01 = disabled (Serial Channel A)
tty02 = disabled (Serial Channel B)
Files Affected: /etc/ttys
To Change: Use the **enable** or **disable** command. (See Appendix B)

C.4.3 Key Values -- sets the values for the erase (destructive backspace) key and the line kill key

Default: erase = ^H (<BACKSPACE>)
kill=^U (<CTRL><U>)
Files Affected: .profile
To Change: Use the **stty** command:
stty erase ^X to set erase
stty kill ^X to set kill
stty <ENTER> to see values
Or edit .profile or /etc/lib/mkuser.prof file.

C.5 Miscellaneous Defaults

Defaults: dump and dumpdir are used by the **sysadmin** program
lpd specifies how many banner pages when a file is printed by **lpr**
mkuser sets the default for home directory and shell that are used by the **mkuser** command

Files Affected: /etc/default/dump
/etc/default/dumpdir
/etc/default/lpd
/etc/default/mkuser

To Change: edit the appropriate file.

Appendix D

Using tsh -- The trsshell Program

The trsshell is a user friendly shell for TRS-XENIX. It allows you to enter commands with easy to remember words instead of letters or abbreviated commands.

To invoke the trsshell type:

```
tsh <ENTER>
```

You can now use any of the trsshell commands described in this appendix as well as the TRS-XENIX commands described throughout this manual.

To exit the trsshell, type:

```
exit <ENTER>
```

You can also type <CTRL><D> to exit tsh.

NOTE: In any tsh command that allow multiple file names to be specified, you may use wildcards.

auto [command line]

The **auto** function creates or removes an automatic command that is executed when tsh is invoked. When an **auto command line** is entered, the command line is stored in the .tshrc file. This file is executed when trsshell starts up. If the file does not exist, it is created.

To cancel the auto function, type **auto** <ENTER> and the automatic command line is removed from the .tshrc file.

backup

To backup a diskette, run the diskutil program at the XENIX Boot prompt (not from within XENIX). To run **diskutil** type:

```
XENIX Boot
:distkutil <ENTER>
```

Further information on **diskutil** can be found in Chapter 8 of this guide.

chdir [directory]

Changes the **current working directory** to the specified directory. If no directory is specified, chdir finds your home directory and makes it the current working directory.

cls

Clears the screen and homes the cursor.

copy [-i] file1 file2
copy [-i] file1 [file2...] directory

-i prompts if the destination file already exists. You then choose to keep or overwrite the existing file.

Copies file1 to file2. The permissions and owner of file2 are kept if the file already exists; otherwise, the permissions of the source file are used.

Multiple files can be copied into a directory by using the second form of copy. The filenames are not changed.

Note: copy refuses to copy a file onto itself.

dir [-abcdgrstuR] [name...]

Lists in long format all files specified. If a directory is specified, the name is printed and its contents are listed.

The listing is sorted alphabetically by default. If no directory name is given, the current directory is listed. By specifying multiple directory names, you can see the contents of several directories. Filenames are listed first followed by the directories when mixed names are listed.

The available options for **dir** are:

-a Lists files beginning with "." as well as all other files. (Super-user mode automatically uses -a.)

-b Prints invisible characters as /nnn (octal)

-c Uses file creation time for sorting or printing.

-d If a directory name is specified, only list its name, not the contents of the directory.

-q Gives group ID instead of owner ID in listing.

-r Reverses the order of the sort to get reverse alphabetic or oldest date first depending on the options specified.

-s Gives size in blocks, including indirect blocks, for each entry.

-t Sorts by time modified.

-u Uses last access time instead of last modification for sorting or printing.

-R Lists the contents of all the subdirectories for the specified directory.

dismount [:] [drive-number]

Informs TRS-XENIX that a previously mounted file system on the specified drive is to be removed.

For a full explanation of the mount process, see "mount".

display [-srw] [-n] [+linenumber] [name ...]

Displays a screenful of text for examination.

It normally pauses after each screenful and prompts "--More--". To see one more line of text, press <ENTER>. To see the next screen of text, press <SPACEBAR>.

The command line options are:

-n an integer number specifying the number of lines for the display window. If this option is omitted, the entire screen is used.

-s "squeezes" multiple blank lines from the output by printing only one blank line. This increases the amount of useful information displayed on the screen.

-r displays control characters as ^c where c represents the character. Otherwise, control character that cannot be interpreted are not displayed.

-w when end of input is reached, "wait" for any key to be pressed before exiting. Otherwise, **display** exits immediately.

+linenumber display file, starting at linenumber.

Display looks in the file /etc/termcap to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

If **display** is reading from a file, rather than a pipe, then a percentage is displayed along with the "--More--" prompt. This gives the fraction of the file (in characters, not lines) that has been read so far.

Other sequences which may be typed when **display** pauses, and their effects are as follows

Note: n is an optional. If omitted, one (1) is used.

n<SPACEBAR> displays i more lines. If n is omitted, the next screenful is displayed.

<CTRL><D> displays 11 more lines ("scroll"). If n is given, then the scroll size is set to n.

d same as ^D (control D)

nz same as typing a space except that n, if specified, becomes the new window size.

ns skips n lines, then print the next screenful of text.

nf skips n screenfuls, then print the next screenful of lines

APPENDIX D

q exits **display** (same as Q)

= displays the current line number.

v starts up the editor vi at the current line.

h Help command; gives a description of all the **display** commands (same as ?).

!command invokes a shell with command. The characters '%' and '!' in "command" are replaced with current file name and the previous shell command respectively. If there is no current file name, '%' is not expanded. The sequences "\!" are replaced by "%" and "!" respectively.

n:n skips to the n-th next file given in the command line (skips to last file if n doesn't make sense)

n:p skips to the n-th previous file given in the command line. If this command is given in the middle of printing out a file, then **display** goes back to the beginning of the file. If n doesn't make sense, **display** skips back to the first file. If **display** is not reading from a file, nothing happens.

:f displays the current file name and line number.

:q or :Q exits from **display** (same as q or Q).

. (dot) repeats the previous command.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command character itself is given, you can press the line kill character (<CTRL><U>) to cancel the numerical argument being formed. In addition, you may press the erase character (<BACKSPACE>) to redisplay the "--More--(xx%)" message.

At any time when output is being sent to the terminal, you may press the quit key (normally <CTRL><7>). **display** will stop sending output, and will display the usual "--More--" prompt.

You may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to **noecho** mode by this program so that the output can be continuous. What you type is not displayed on your terminal, except for the slash (/) and exclamation (!) commands.

do *filename*

The list of commands stored in *filename* are executed as keyboard entries. Nesting of *do* commands is allowed as deep as is necessary or as the system has memory.

files [-abcdqrstux1CR] [*name*...]

Lists in columns all files in the specified directory.

The listing is sorted alphabetically by default. If no name is given, the current directory is listed. By specifying multiple directory names, you can see the contents of several directories. File names are listed first, then the directories when mixed names are specified.

Also, all directories are marked with a trailing "/" and executable files are marked with a trailing "*".

The available options for **files** are:

-a Lists files beginning with "." as well as all other files. (Super-user mode defaults to -a.)

-b Prints invisible characters as /nnn (octal)

-c Uses file creation time for sorting and printing.

-d If a directory name is specified, only list its name, not the contents of the directory.

-q Gives group ID instead of owner ID in listing.

-r Reverses the order of the sort to get reverse alphabetic or oldest date first depending on the options specified.

-s Give size in blocks, including indirect blocks, for each entry.

-t Sorts by time modified.

-u Uses last access time instead of last modification for sorting and printing.

-R Lists the contents of all the subdirectories for the specified directory.

-x Forces columnar printing (Sorted material is printed across rather than down the page.)

-1 Forces a one entry per line output format, e.g. to a teletype.

-C Forces multi-column output, this is used with output redirection.

free [filesystem...]

Prints out the number of free blocks available on the specified filesystem. If no file system is specified, the free space on all of the normally mounted file systems is printed.

help [subject]

Displays the help text for the specified subject. A few special help commands are:

help *	displays a list of subjects for which help is available.
help	displays an explanation of the help text and the notations used within.

If the help text is longer than one screen, the message --More--(nn%) is displayed. Press <SPACEBAR> to see the next screenful, or press <ENTER> to expose one more line. To abort the display of help text, press <q>.

i

To inform the system you have placed a diskette in a drive or wish to remove one, you need to use the commands mount and dismount. If you have placed a XENIX disk in drive Ø, for example, you may type "mount Ø" to tell the system to make its contents available from the directory "/mntØ". Likewise, when you are finished and wish to remove it, you type "dismount Ø" and then the system unmounts your floppy.

kill [-r] file...

-r Deletes the entire contents of the specified directory, and the directory itself.

Removes the entries for one or more files from a directory, thus destroying the file. A directory must have write permission if files are to be deleted from it. The file however, does not need to have read or write permission to be deleted.

You are prompted for each file before it is removed. Answer <y> <ENTER> to delete the file.

If a designated file is a directory, an error comment is printed unless the optional argument -r has been used.

You cannot remove the file ".." from a directory.

lib

Displays a list of the trsshell commands.

mount drive-number

Announces to the system that a file system is now present on Drive drive-number. The filesystem's contents are made available in the directory "/mntn", where n is the specified drive. For example, if "mount Ø" is typed, then the contents of the floppy on drive Ø is made available in the directory "/mntØ".

Dismount performs the inverse operation, announcing to the system that the file system previously mounted on Drive drive-number is to be removed.

Mount will refuse to mount a file system which is not marked clean: this can happen if a system crash prevented the use of dismount or /etc/shutdown. In such a case, use fsck to clean the file system, then try mount again. (See Chapter 7 of this manual for more information.)

These commands keep track of where each device is mounted. If no drive-number is given, mount prints a table showing the currently mounted filesystems (if any).

Note: you cannot mount a disk with an alien filesystem on it, for example, one with TRSDOS on it. If you attempt to do so, you will crash the system. Likewise, you cannot mount a floppy or hard disk which has been newly formatted. You must use the mkfs utility on it first.

move [-i] file1 file2
move [-i] file1 [file2...] **directory**

-i prompts if the destination file already exists. You then choose to keep or overwrite the existing file.

Copies file1 to file2. The permissions and owner of file2 are kept if the file already exists; otherwise, the permissions of the source file are used.

Multiple files can be copied into a directory by using the second form of move. The filenames are not changed.

Note: move refuses to copy a file onto itself.

print [-r|-c] [-m|-n] [file...]

Causes the file(s) to be queued for printing on the line printer. If no files are named, the keyboard is read until you type a ^D at the beginning of a line.

The following options are available:

-r Removes the file when it has been queued.

-c Copies the file to avoid any changes being made before it's printed.

-m Reports by mail when printing is complete.

-n Does not report by mail. This is the default option.

rename old.file new.filename

Changes old.file's name to new.filename.

restore [:] drive-number [-d] | [name...]

-d displays a list of files on the floppy diskette (directory).

Reads in files that have been written to a floppy with **save** and restores them to where they were saved from. You may use multiple filenames or directory names. If a directory name is specified, it restores the directory and it's files and subdirectories (and their contents).

save [:] drive-number {-ss|-ds} [name...]

-ss specifies that the destination floppy is a single-sided disk.
-ds specifies that the destination floppy is a double-sided disk.

One of the above (-ss, -ds) must be specified.

Saves files or entire directories and their contents to the floppy in the specified drive. You may use multiple file names or directory names. If a directory name is specified, it saves the directory and it's files and subdirectories (and their contents).

You must tell save if you are using single sided diskettes or double sided by using the -ss flag if they are single sided, and by using the -ds flag if they are double sided. The floppy disks must be formatted prior to use.

set [option]

Allows you to change tsh's internal options. To display the options settings, type **set** <ENTER>.

Options that are currently available are:

verbose [on|off] -- if verbose is off, trsshell's does not inform you of a command's success or failure.

retry [on|off] -- enables/disables the retry feature. When retry is on, the command line is converted to lower-case if trsshell was unable to execute a command. Trsshell then tries to execute the lower-case command line.

prompt string -- sets the trsshell prompt to string.
For example: set prompt cue> <ENTER>, sets the string "cue>" as the new trsshell prompt.

showdir

Prints the pathname of the current working directory.

time

Displays the current date and time.

version

Displays the name of the shell and its current version.

APPENDIX E

Installing TRS-XENIX Applications and How to Use save To Make Backups

Installing TRS-XENIX Applications

Before installing your TRS-XENIX Applications package please read through all the directions. This will enable you to become familiar with the entire procedure, and help make the installation smoother. During the installation be sure to watch the screen and answer all the prompts you are given.

To begin installing the package:

1. Turn on all peripherals and then turn on the Computer.

2. Login as root (to install TRS-XENIX applications, you must be logged in as root).

3. At the root prompt (#) type:

   install <ENTER>

   The screen will display a TRS8Ø Model-16 XENIX System, Application Installation Menu. It will also display the prompts 1) to install or q) to quit.

4. Enter <1> to install your application.

5. At the prompt "Insert" your TRS-XENIX Application diskette in Drive Ø and press <ENTER>.

The next screen will display the name of the application being installed, the version number, and the catalog number. At the bottom of the screen a welcoming message will appear.

While your application is being installed, the system also creates the file (/etc/logbook) that TRS-XENIX updates whenever a new application is installed or updates are made.

When XENIX is finished the message:

"Installation complete - Remove the diskette, then press <ENTER>"

will appear. Pressing <ENTER> returns the menu to the screen. You are then prompted to press:

1) to install another application, or

q) to quit.

Your installation is now complete.

You may now install another TRS-XENIX application or quit. (Pressing <q> returns you to TRS-XENIX.)

Making Backups Using save

Use the following procedure to make backups instead of what is described in your applications manual. Before getting started, you must format some floppy disks. You may want to keep a box of floppy disks formatted to make backups with.

Instructions for formatting floppy disks are given in Chapter 8.

> **HINT:** Since you must exit TRS-XENIX to format floppy disks, you may want to format them at the end of the day after the system is shutdown (or at the begining of the day).

After you have formatted plenty of floppy disks, follow this procedure to make backups with the **save** command:

1. Boot TRS-XENIX and log in as root, while you are at the console in multi-user mode. At the root prompt, type:

   save <ENTER>

2. The following menu is displayed:

```
Save/Restore User File Systems
------------------------------
1  to do full daily data save
2  to do daily data save by system
3  to restore full daily data save
4  to restore daily data save by system
5  to save programs
6  to restore programs
q  to quit
```

3. You are now prompted to enter a selection. A description of each selection follows:

   full daily data save (1) -- saves the data files for all TRS-XENIX Applications on the hard disk.

   daily data save by system (2) -- saves the data files for one TRS-XENIX Application. You are prompted to enter the two letter initial of the system (i.e. Ar, Gl, Ap).

   restore full daily data save (3) -- restores files saved by selection (1).

   restore daily data save by system (4) -- restores files saved by selection (2).

   save programs (5) -- Saves all TRS-XENIX Application programs.

   restore programs (6) -- restores all TRS-XENIX Application programs, stored with selection (5).

4. After you enter your selection, you are prompted to enter a <1> if you are using single-sided diskettes or a <2> if you are using double-sided diskettes.

5. You are now prompted:

   Enter Drive Number:

   Enter a <Ø> or a <1> for Floppy Drive Ø or Floppy Drive 1 respectively.

6. Some selections prompt you for more information. You are then prompted to insert the first floppy disk into the drive. When the first floppy disk is full, you are prompted to enter the next disk, and so on.

7. You should label each floppy disk with a description, date and disk number in the backup series:

   General Ledger Data Files ** BACKUP **
   January 23, 1983 Disk # 1 of 1Ø

8. When the process is complete, you are returned to the SAVE menu. If you are through, press <q> to exit to the root prompt.

Appendix F

Trasferring TRSDOS-II FIles to TRS-XENIX

The **tx** command is used to transfer files from TRSDOS-II formatted diskettes to TRS-XENIX. It also transfers files from TRSDOS-II SAVE formatted diskettes (-r option).

When the files are transferred to TRS-XENIX, their filenames are changed to meet XENIX file specifications. That is all uppercase letters are changed to lowercase, and the slash (/) preceding the extension is changed to a period (.).

The **tx** command also allows you to display directory information for TRSDOS-II diskette.

Notes:

1. Password protected files are transferred to TRS-XENIX. Ownership is given to the user only.

2. The /dev files for the floppy drives must have the "r" permission set (/dev/rfdn and /dev/fdbtn -- n is Ø,1,2 or 3 depending on the drive number).

Important: TRS-XENIX can only read TRSDOS-II 4.x media. If you wish to transfer TRSDOS files to TRS-XENIX, use the FCOPY command to copy the files to TRSDOS-II and then use the **tx** command to transfer the files to TRS-XENIX.

The syntax for **tx** is:

a) **tx** :d options Tfspec|Twc Xdir

b) **tx** :d options Tfspec Xfspec

c) **tx** :d -f [Tfspec|Twc]

options can be any of the following:

-a absolute, copy files even if file already exists
-c do not convert upper case letters in the filename to lower case
-l do not add a line feed to TRSDOS-II VLR files
-p prompt operator before each file is moved
-r specifies that the source is TRSDOS-II SAVE media
-s do not strip length byte from TRSDOS-II VLR files
-v verbose, display informative messages during copy.
-x converts carriage returns to linefeeds (ØAH to ØDH)

Where:

Tfspec	=	TRSDOS-II file specification
Twc	=	TRSDOS-II wildcard specification
Xfspec	=	TRS-XENIX file specification
Xdir	=	TRS-XENIX directory
:d	=	specifies the drive that contains the TRSDOS-II diskette. :d is required

If Xdir is omitted, the default is the current working directory (form a of syntax).

The options and drive number (:d) are position independant. They may be located in any order on the command line (except within filespecs).

If a wildcard mask is used, it must be enclosed in quotes.

Examples

1. tx :Ø

   Copies all files from the TRSDOS-II diskette in Drive Ø to the current working directory of TRS-XENIX (default).

2. tx :Ø ARMAIN/DAT .
   tx ARMAIN/DAT . :Ø

   Copy file ARMAIN/DAT from the TRSDOS-II diskette in Drive Ø to the current working directory (.). The TRS-XENIX filespec is **armain.dat.**

3. tx :1 LEAOWN/IDX LEAOWN/DAT OWNER/DAT /usr/lease/owners -x
   tx -x LEAOWN/IDX LEAOWN/DAT OWNER/DAT /usr/lease/owners :1
   tx LEAOWN/IDX :1 -x LEAOWN/DAT OWNER/DAT /usr/lease/owners

   Copies files: LEAOWN/IDX, LEAOWN/DAT and OWNER/DAT from the TRSDOS-II diskette in Drive 1 to the TRS-XENIX directory /usr/lease/owners. The TRS-XENIX filenames are: leaown.idx, leaown.dat, and owner.dat.

   Any carriage returns in the files are converted to line feed. This is useful when transferring BASIC source to TRS-XENIX.

4. tx :1 -c LEAOWN/IDX LEAOWN/DAT OWNER/DAT /usr/lease/owners
   tx LEAOWN/IDX LEAOWN/DAT OWNER/DAT -c /usr/lease/owners :1
   tx LEAOWN/IDX -c :1 LEAOWN/DAT OWNER/DAT /usr/lease/owners

   Copies files: LEAOWN/IDX, LEAOWN/DAT and OWNER/DAT from the TRSDOS-II diskette in Drive 1 to the TRS-XENIX directory /usr/lease/owners. Since the -c switch was specified, the filenames are not converted to lowercase:

TRSDOS-II	TRS-XENIX
LEAOWN/IDX	LEAOWN.IDX
LEAOWN/DAT	LEAOWN.DAT
OWNER/DAT	OWNER.DAT

5. tx :3 "AR*" /usr/AR -pav
   tx -p -a -v :3 "AR*" /usr/AR
   tx "AR*" /usr/AR :3 -apv

   Displays all files found on the TRSDOS-II diskette in Drive 3, that match the wildcard mask **AR***. The -p option causes the message **copy ?** to be displayed. Answer <y>es or <n>o. If <y> is entered, the TRSDOS-II file is copied into the TRS-XENIX directory /use/AR.

   Because the absolute switch was specified (-a), you are not prompted if the destination file already exists. However, informative messages are displayed because the -v switch was specified.

6. tx :Ø -f

   Displays directory information for every file on the TRSDOS-II diskette in Drive Ø.

7. tx :1 -f -r

   Displays directory information for every file on the TRSDOS-II SAVE diskette in Drive Ø.

8. tx :3 INIT/* -f

   Displays directory information for all files matching the wildcard mask INIT/* found on the TRSDOS-II diskette in Drive 3.

9. Wildcard Example:

The wildcard character "*" matches zero or more occurrences of any legal filespec character including the slash.

The wildcard character "?" matches only one occurrence of any legal filespec character including slash.

Suppose you have a diskette with the following files in the directory:

TEST	TEST1	TEST/DAT
TESTA/DAT	TEST1/DAT	TEST/C

The wild card mask:

"TEST*"

matches all of the above files. It is important to remember that TRS-XENIX does not follow TRSDOS-II conventions for wildcarding. TRS-XENIX does not recognize extensions, therefore a TRSDOS-II extension is considered to be part of the filename. In this example "TEST*" matches **any** filename that begins with the letters "TEST",

However the wildcard mask:

"TEST?"

matches only the file TEST1 because the "?" character means to match only one letter. Therefore filespecs with extensions are too long to match this wildcard.

The wildcard mask:

"TEST?*"

matches the following files:

TEST1	TEST/DAT	TESTA/DAT
TEST1/DAT	TEST/C	

The file TEST did not match because the "?" character must match at least one character.

Note: Any filespec containing the wildcard mask characters (* or ?) must be enclosed in quotes.

ERRORS

"Unknown option found" An illegal option was used.

"Multiple drive specification" More than one device type was specified.

"Illegal drive unit number" An illegal drive number was given. Drive numbers may be in the range [Ø,7].

"Drive number not found on the command line" A source drive number is required in the command line.

"Directory Xdir does not exist" The directory name specified was not found on the TRS-XENIX file system.

"Filestat failed on Tfspec" Internal Error. The specified file was opened successfully, but the file status could not be read.

"Filename Xdir is not a directory" The TRS-XENIX directory specified is not a directory, but is a file.

Appendix W: Basic Concepts

This appendix gives you an understanding of the basic concepts that you need to function in the XENIX environment. It discusses the XENIX file system, naming conventions, commands, and input and output. After reading this appendix you will have an understanding of how the system's files, directories, and devices are organized and named, how commands are entered, and how a command's input and output can be manipulated.

W.1 File System

A file system is a set of files organized in a logical fashion. In XENIX, this set of files consists of all available system resources including data files, programs, line printers, and disks. Thus, the XENIX file system is more generally a system for accessing all of the resources of the system.

To logically structure the files and resources of the system, the XENIX file system is organized hierarchically into a "tree-structure." See Figure W-1 for an illustration of a typical tree-structured file system. In this tree of files, the root of the tree is at the top and branches of the tree grow downward. Directories correspond to nodes in the tree; ordinary files correspond to "leaves." If a directory contains a downward branch to other files or directories, then those files and directories are "contained" in the given directory. It is possible to specify any file in the system by starting at the root (where the root is at the top) and traveling down any of the branches to the desired file. Similarly, you can specify any file in the system, relative to any other. Specification of these files depends on a knowledge of XENIX naming conventions, discussed in Section W.3.

Privacy and security for files and directories can be arranged. Each file and directory has read-write-execute permissions that can be set to control access by the owner, by a group of users, and by everyone else.

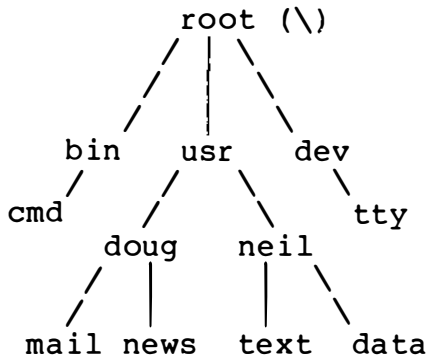


Figure W-1. A Typical File System

In the typical tree-structured file system of Figure W-1, the "tree" grows downward. The names root, bin, usr, dev, doug, and neil all represent directories, and are all nodes in the tree. By convention, in XENIX the name of the root directory is given the one character name, "/". The names mail, news, text, and data all represent normal data files, and are all "leaves" of the tree. Note that cmd, which here is an executable command, is also a leaf and therefore a file. The name tty represents a terminal and is also represented in the tree.

W.1.1 Files

The file is the fundamental unit of the file system. Conceptually, everything is treated as a file. However, there are really three different type of files: ordinary files (what we usually mean when we say "file"), special files, and directories. Each of these file types is discussed below:

Ordinary Files

Ordinary files typically contain textual information such as documents, data, or program sources. Executable binary files are also of this type. Any file that is not a special file or a directory is an ordinary file.

Special Files

A special file is one that corresponds to a physical device of some sort, such as a disk, a line printer, a terminal, or system memory. To the XENIX user, special files can be treated like ordinary files. However, the internal handling of a printer or terminal is much different from that of an ordinary disk file, and the operations that can be performed on devices vary from those that

can be performed on ordinary files.

Directories

Directories are read-only files containing information about the files or directories that are conceptually (but not physically) contained within them. The nesting of directories in other directories is the way in which XENIX implements its characteristic tree-structured directory system. Directories are discussed further in the next section.

W.1.2 Directory Structure

With multiple users and multiple projects, the number of files in a file system can proliferate rapidly. Fortunately, as explained earlier, XENIX organizes all files into a tree-structured directory hierarchy. Each user of the system has his own personal directory. Within that directory, the user may have directories or other subdirectories owned and controlled only by the user.

When you log in to XENIX, you are "in" your directory. Unless you take special action when you create a file, the new file is created in your working directory. This file is unrelated to any other file of the same name in someone else's directory.

A diagram of part of a typical user directory is shown in Figure W-2.

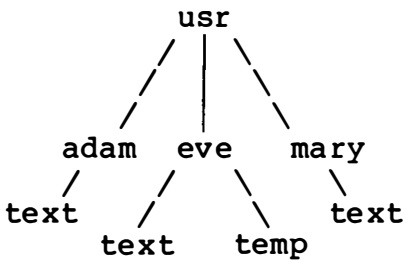


Figure W-2. A Typical User Directory

In Figure W-2, the usr directory contains each user's own personal directory. Notice that Mary's file named text is unrelated to Eve's. This is not important if all the files of interest are in Eve's directory, but if Eve and Mary work together, or if they work on separate but related projects, this division of files becomes handy indeed. For example, Mary could print Eve's text by typing:

```
pr /usr/eve/text
```

Similarly, Eve could find out what files Mary has by typing:

```
ls /usr/mary/*
```

W.2 Naming Conventions

Now that we have discussed what the XENIX file system is, and what it consists of, we need to use filenames in a more precise way. The first thing to remember is that every single file, directory, and device in XENIX has both a filename and an absolute pathname. The absolute pathname is unique to all names in the system; filenames are unique only within directories and need not be unique system-wide. This is similar to someone who's "absolute" name is John Robert Smith, but whom everyone calls John. The name John need not be unique, although it will greatly simplify life if John Robert Smith is a unique name.

W.2.1 Filenames

A simple filename is a sequence of 1-14 characters other than a slash (/). Every single file, directory, and device in the system has a filename. Filenames are used to uniquely identify directory contents. Thus, no two names in a directory may be the same. However, filenames in different directories may be identical.

W.2.2 Pathnames

A pathname is a sequence of directory names followed by a simple filename, each separated from the previous one by a slash. If a pathname begins with a slash, the search for the file begins at the root of the entire tree. Otherwise, it begins at the user's current directory (also known as the working directory). A pathname beginning with a slash is called a full (or absolute) pathname because it does not vary with regard to the user's current directory. A pathname not beginning with a slash is often called a relative pathname, because it specifies a path relative to the current directory. The user may change the current directory at any time by using the cd command.

In most cases, a filename and its corresponding pathname may be used interchangeably.

W.2.3 Sample Names

Some sample names follow:

/	The absolute pathname of the root directory of the entire file system.
/bin	The directory containing most of the frequently used XENIX commands.
/usr	The directory containing each user's personal directory. The subdirectory, /usr/bin contains frequently used XENIX commands not in /bin.
/dev	The directory containing files corresponding to each available physical device (e.g., terminals, line printers, and disks).
/lib	The directory containing special data files used by some standard commands.
/tmp	This directory contains temporary scratch files.
/usr/joe/project/A	This is a typical full pathname. This one happens to be a file named A in the directory named project belonging to the user named joe.
bin/x	A relative pathname; it names the file x in subdirectory bin of the current working directory. If the current directory is /, it names /bin/x. If the current directory is /usr/joe, it names /usr/joe/bin/x.
file1	Name of an ordinary file in the current directory.

Each user resides "in" a directory called the current directory. All files and directories have a "parent" directory. This directory is the one immediately above and "containing" the given file or directory. The XENIX file system provides special shorthand notations for this directory and for the current directory:

- . The shorthand name of the current directory. Thus ./filexxx names the same file as filexxx, if such a file exists in the current directory.

.. The shorthand name of the current directory's parent directory. If you type

```
cd ..
```

then the parent directory of your current working directory becomes your new current directory.

Although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. For instance, the dash (-) is used in specifying command options, and should be avoided when naming files. To avoid pitfalls, you will do well to use only letters, numbers, and the period.

W.2.4 Special Characters

XENIX provides a pattern-matching facility for specifying sets of filenames that match particular patterns. For example, examine the problem that occurs when naming the parts of a large document, such as a book. Logically, it can be divided into many small pieces: chapters or perhaps sections. Physically, it must be divided too, since the XENIX editor, ed, cannot handle really big files. Thus, you should construct a document as a number of files. For example, you might have a separate file for each chapter:

```
chap1
chap2
...
```

Or, if each chapter were broken into several files, you might have:

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

You could then tell at a glance where a particular file fits into the whole.

There are other advantages to a systematic naming convention that are not so obvious. What if you wanted to print the whole book? You could type

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

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut: a sequence of similar names can be specified with the use of two special "wild card" characters.

For example, you can type:

```
pr chap*
```

The asterisk (*), called "star" in XENIX, means "zero or more characters of any type," so this translates into "print all files whose names begin with the word "chap", listed in alphabetical order."

This shorthand notation is not a unique property of the pr command; it is a system-wide service of the shell program that interprets commands, sh. Using this fact, you list the names of the files in the book by typing:

```
ls chap*
```

This produces

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

The star is not limited to the last position in a filename; it can be used anywhere and can occur several times. As a special case, a star by itself matches every filename, so

```
rm *
```

removes all your files.

The star is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4, and 9. Then you can say

```
pr chap[12349]*
```

The brackets ([and]) mean "match any of the characters inside the brackets." A range of consecutive letters or digits can be abbreviated, so you can also do this with

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

(Note that this does not match forty-nine filenames, but

only five.) Letters can also be used within brackets: "[a-z]" matches any character in the range "a" through "z".

The question mark (?) matches any single character, so

```
ls ?
```

lists all files that have single-character names, and

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

lists information about the first file of each chapter (i.e., chap1.1, chap2.1, ...).

Of these pattern matching conventions, the star (*) is certainly the most useful, and you should get used to it at once.

If you should ever need to turn off the special meaning of any of the special characters (*, ?, and [...]) enclose the entire argument in single quotes. For example, the following command will print out only files named "?" rather than all one character filenames:

```
ls '?'
```

W.3 Commands

Commands are used to invoke executable programs. When you type the name of a command, the XENIX shell looks for a program with the given name to execute. If the shell finds an executable program, it will execute it. Commands always contain the name of the executable program as the first word of the command. Commands may also contain switches that specify options or other arguments as needed by the program. Commands are entered on a command line that is read by the shell. Command lines are discussed in the following subsection.

W.3.1 Command Line

Whether typing at the terminal, or executing commands from a file, XENIX always reads commands from command lines. The command line is a line of characters that is scanned and read by the shell command interpreter to determine what to do next. When you are typing at a terminal, you are editing a line of text called the command-line buffer that becomes a command line only when you type <RETURN>. This command-line buffer can be edited with the <BKSP> and <CONTROL-U> keys. Typing <RETURN> causes the command-line buffer to be submitted to the shell as a command line. The shell reads

the command line and executes the appropriate command. If you type <INTERRUPT> before you type <RETURN>, then the command-line buffer is aborted. Multiple commands can be entered on a single command line so long as they are separated by a semicolon (;). For example, the following prints out the current date and the name of the current working directory:

```
date ; pwd
```

Commands can be submitted for processing in the background by appending an ampersand to the command. Thus

```
mv file1 file2 file3 file4 dir1&
```

will move the files file1, file2, file3, and file4 to the directory dir1 without tying up your terminal. You can execute other commands from your terminal in the foreground while the **mv** command executes in the background.

W.3.2 Syntax

The general syntax for commands is as follows:

```
cmd  [ switches ][ arguments ] [ filenames ]
```

In practically all cases, command names are all lowercase. Switches are flags that select various options available when executing the command. Switches are optional and always precede other arguments and filenames. Switches consist of a dash prefix (-) and an identifying alphanumeric character. Some switches are also prefixed by a plus sign (+). Switches can often be grouped as a single switch as in:

```
ls -arl
```

Here the -a switch (pronounced "minus a") selects the option which lists all files in the directory. The -r switch selects the option which causes the names in the directory to be sorted in reverse alphabetical order. And the -l switch selects the option which causes listing of a long format for each directory entry.

Sometimes switches must be given separately, as in:

```
copy -v -a source dest
```

Here the -v switch specifies a verbose option. The -a switch tells the copy command to ask the user before copying the two given directories.

Arguments of various types can also be given, such as search strings, as in:

```
grep 'string of text' outfile
```

In the above example, "string of text" is an argument and is the search string that the grep command searches for in the file outfile. Outfile, itself, is a filename argument that specifies the name of a file required by the command.

In most cases, commands are executable object files compiled from C programs. In some cases, commands are executable command files called "shell procedures."

W.4 Input and Output

XENIX handles input and output from commands in a unique way: it assumes that input and output, by default, are associated with the terminal from which the command originates. That is, input comes from the keyboard and output goes to the terminal screen. To illustrate typical command input and output, type:

```
cat
```

This command now expects input from your keyboard. It will accept as many lines of text as you can type as input, until you type a <CONTROL-D> as an end-of-file indicator. For example, type:

```
this is two lines
of input
<CONTROL-D>
```

When you type the <CONTROL-D>, input ends and output begins. The cat command then immediately outputs the two lines that you typed -- since output is sent to the terminal screen by default, that is where the two lines are sent. Thus, the complete session will look like this on your terminal screen:

```
$ cat
this is two lines
of input
this is two lines
of input
$
```

The flow of command input and output can be "redirected" so that input comes from a file instead of from the terminal keyboard, and so that output goes to a file or to a line

printer, instead of to the terminal screen. In addition, "pipes" can be created that allow the output from one command to become the input to another. Redirection and pipes are the subjects of the next two subsections.

W.4.1 Redirection

It is universal in XENIX systems that a file can replace the terminal for either input or output. For example

```
ls
```

displays a list of files on your terminal screen. But if you say

```
ls >filelist
```

a list of your files is placed in the file filelist (which is created if it does not exist). The output redirection symbol (>) means "put the output from the command into the following file, rather than display it on the terminal screen." As another example, you could combine several files into one by capturing the output of cat in a file:

```
cat f1 f2 f3 >temp
```

The output append symbol (>>) operates very much like the output redirection symbol (>) does, except that it means "add to the end of." That is

```
cat file1 file2 file3 >>temp
```

means to concatenate file1, file2, and file3 to the end of whatever is already in temp, instead of overwriting and destroying the existing contents. As with normal output redirection, if temp doesn't exist, it is created for you.

In a similar way, the input redirection symbol (<) means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of editing commands and put them into a file called script. Then you could execute the commands in the script on a file using the XENIX editor by typing:

```
ed file <script
```

As another example, you could use ed to prepare a letter in file letter.txt, then send it to several people with

```
mail adam eve mary joe <letter.txt
```

W.4.2 Pipes

One of the major innovations of the **XENIX** system is the concept of a pipe. A pipe is simply a way to connect the output of one command to the input of another command, so that the two run as a sequence of commands called a pipeline.

For example

```
pr frank.id george.id hank.id
```

prints the files named frank.id, george.id, and hank.id, beginning each on a new page. Suppose you want them run together instead. You could type:

```
cat frank.id george.id hank.id >temp
pr <temp
rm temp
```

But this is more work than is necessary. What we want is to take the output of cat and connect it to the input of pr. So we use a pipe:

```
cat frank.id george.id hank.id | pr
```

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

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program **wc** counts the number of lines, words, and characters in its input, and **who** prints a list of currently logged on people, one per line. Thus,

```
who | wc
```

tells how many people are logged on. And of course

```
ls | wc
```

counts your files.

Any program that reads from the terminal keyboard can read from a pipe instead. Any program that displays output to the terminal screen can send input to a pipe. You can have as many elements in a pipeline as you wish.

Many XENIX programs are written so that they take their input from one or more files, if file arguments are given. If no arguments are given, they read from the terminal keyboard, and thus can be used in pipelines. For example

```
pr -3 albert.txt bernard.txt carl.txt
```

prints, in order, the files albert.txt, bernard.txt, and carl.txt. But in

```
cat albert.txt bernard.txt carl.txt | pr
```

pr prints the concatenation of these files coming down the pipeline. The difference is that here, albert.txt, bernard.txt, and carl.txt are run together and then treated as one file rather than three.

Appendix X: Frequently Used Commands

X.1 The Model-16 Console Keyboard

In the description of the commands in this appendix, note the mapping of Model-16 console keys to important XENIX characters not normally available on the standard Model-16 console keyboard:

XENIX Character	Model-16 Console
<INTERRUPT>	<BREAK>
<INTERRUPT>	<CONTROL-C>
<CONTROL-S>	<HOLD>
\ (backslash)	<CONTROL-/>
` (back quote)	<CONTROL-'>
| (vertical bar)	<CONTROL-!>
	<CONTROL-1>

X.2 Gaining Access to the System

Logging In

To gain access to the system, respond to the "login:" prompt by typing your user name followed by a <RETURN>. Then respond to the password: prompt with your password. For example, a login for the user joe might look like this:

```
login:joe
password:abracadabra
```

Note that the password is not shown on the terminal screen.

Logging Out

The logout procedure is simple -- all you need to do is type:

```
<CONTROL-D>
```

Since within other programs, <CONTROL-D> signifies the end-of-file, at times it may be necessary to type <CONTROL-D> several times before you can log yourself out.

Changing Your Password

To change your password, use the **passwd** command. For the user joe, a session might go like this:

```
Changing password for joe
Old password: palooka
New password: Bazookah
Retype new password: Bazookah
```

To maintain security, none of your responses are shown on the screen. It is best to mix upper- and lowercase letters and to pick a password greater than five characters in length. These measures should be taken to foil automated attempts at guessing your password.

X.3 Terminal Configuration

Setting Terminal Options

There are a number of terminal options that can be set with the command stty. When entered without parameters, **stty** displays the current terminal settings. For example, typical output might look like this:

```
speed 9600 baud
erase '^h' ; kill '^u'
even -nl
```

This says that the rate of data transmission to and from the terminal is 9600 baud, that the backspace character (erase) is <CONTROL-H>, that the line kill character is <CONTROL-U> that even parity is set, and that a <RETURN> is acceptable as a new line character. Each of the above characteristics can be set with **stty**.

Changing Terminals

If you have to log in to XENIX on a terminal of a type different than the terminal you normally use, you may need to change the shell TERM variable. This is normally set to the proper default terminal when you login, but if you switch terminals, you'll need to type something like:

```
TERM=termname; export TERM
```

where termname is the name of a known terminal. A wide variety of terminals are supported; terminal names are listed in the system file named /etc/termcap.

X.4 Status Information

Finding Out Who is on the System

The **who** command lists the names, terminal line numbers, and login times of all users currently logged on the system. For example, type:

```
who
```

This command should produce something like the following output on your terminal screen:

```
arnold   tty02   Apr  7 10:02
daphne   tty21   Apr  7 07:47
elliot   tty23   Apr  7 14:21
ellen    tty25   Apr  7 08:36
gus      tty26   Apr  7 09:55
adrian   tty28   Apr  7 14:21
```

Finding out What Processes are Running

Because processes can be placed in the background for processing, it is not always obvious which processes you are responsible for. The **ps** command stands for "process status" and lists information about currently running processes associated with your terminal. For instance, the output from a **ps** command might look like this:

```
  PID TTY TIME CMD
10308 38  1:36 ed chap02.man
   49 38  0:29 -sh
11267 38  0:00 sh -c ps
```

The PID column gives a unique process identification number that can be used to kill a particular process. The TTY column gives the terminal that the process is associated with. The TIME column gives the cumulative execution time for the process.

X.5 Process Control and Command Line Editing

Placing A Process in the Background

Normal commands executed at the keyboard are executed in strict sequence: one must finish executing before the next can begin. Executing commands of this type are called foreground processes. A background process, in contrast, need not finish executing before you execute your next command. Background commands are

especially useful for commands that may take several minutes or even hours to complete, because they can be placed in the background while you continue executing other commands at your terminal.

To place a process in the background, type an ampersand (&) at the end of the command. For example, to move files from the current directory to another directory, while simultaneously continuing with whatever else you have to do, type:

```
mv file1 file2 file3 otherdir&
```

Note that when processes are placed in the background, you lose control of them as they execute. For instance, typing <INTERRUPT> does not abort a background process. You must use the **kill** command instead, described below.

Killing a Process

To abort execution of a foreground process press your terminal's <INTERRUPT> key. This kills whatever foreground command you have running. To kill all of your processes that are executing in the background, type:

```
kill 0
```

To kill only a specified process executing in the background, first type:

```
ps
```

Ps displays the Process Identification Numbers (PIDs) of your existing processes:

```
  PID TTY TIME CMD
 3459 03  0:15 -sh
 4831 03  1:52 ed chap01.s
 5185 03  0:00 sh -c ps
```

Next, you might type

```
kill 4831
```

where 4831 is the PID of the process that you want killed.

Erasing A Command Line

When entering commands, typing errors will occur. To erase the current command line so that you can start retyping a new one, enter a <CONTROL-U>, as shown below:

```
kat file2<CONTROL-U>
cat filel
```

In the above command line, the first line is aborted and automatically a newline is generated so that typing may resume. You then can enter the correct command line.

Halting Screen Output

In many cases, you will be examining the contents of a file on the terminal screen. For longer files, the contents will often scroll off the screen faster than you can examine them. To temporarily halt a program's output to the terminal screen, type <CONTROL-S>. To resume output, type any key except <INTERRUPT>.

X.6 File Manipulation

Creating Files

To create an empty file, simply type:

```
>filename
```

Here, filename is the name of the newly created file. The greater-than sign (>) is used to redirect output from the terminal to a file. In this special case no information is sent. In general, new files are created by commands as needed.

Displaying File Contents

To display the contents of a file, use the cat command. **Cat** displays the contents of a file on the default standard output file which is the terminal screen. For example, the following command displays the contents of filel on the screen:

```
cat filel
```

Cat can also display the contents of more than one file as in

```
cat file1 file2
```

Combining Files

Cat, as mentioned above, is normally used to send the contents of files to the terminal screen. However, the name **cat** comes from the word concatenate, and **cat** is also frequently used to combine files into some other new file. Thus, to combine the two files named file1 and file2, and to create a new file named bigfile, type:

```
cat file1 file2 >bigfile
```

Note here that we are putting the contents of the two files into a new file with the name bigfile. The greater than sign (>) is used to redirect normal output of the **cat** command from the terminal screen to the new file.

Moving a File

With the **mv** command, two ways of moving a file are supported:

1. Moving a file so that it now has a new name. For instance, to move a file named text to a new file named book, type:

   ```
   mv text book
   ```

   After this move completes, no file named text will exist in the working directory.

2. Moving a file into a specified directory. In this case, you give the name of the destination directory as the final name in the move command. For instance, to move file1 and file2 into the directory named /tmp, type:

   ```
   mv file1 file2 /tmp
   ```

   The two files you have moved no longer exist in your working directory, but files with the same names now exist in the directory /tmp. The above command has exactly the same effect as typing the following two commands:

   ```
   mv file1 /tmp/file1
   mv file2 /tmp/file2
   ```

Remember that the **mv** command always checks to see if the last argument is the name of a directory, and, if so, all files designated by filename arguments are moved into that directory.

Renaming a File

To rename a file, you simply "move" the file to a file with the new name: the old name of the file is removed. Thus, to rename the file anon to johndoe, type:

```
mv  anon johndoe
```

Note that moving and renaming a file are essentially identical operations.

Copying a File

There are two forms of the cp command: one in which a file is copied to another file and another in which files are copied into a directory. Thus, to create two copies of a file in your own working directory, you must rename the new copy. To do this, the copy command can be invoked as follows:

```
cp file clone-of-file
```

After the above command has executed, two files with identical contents reside in the working directory.

To copy three files into a directory named filedir, type:

```
cp file1 file2 file3 filedir
```

In the above command, three files are copied into the directory filedir; the original versions still reside in the working directory. There is a one-to-one correspondence between the names in the two directories.

Deleting A File

To delete or remove files, simply type:

```
rm file1 file2
```

In the above command, the files file1 and file2 are removed from your working directory.

X.7 Editing

The following commands all deal with use of the XENIX line editor, **ed**.

Invoking the Editor

To invoke **ed**, type:

ed filename

where filename is the name of the file you want to edit. If no name is given, a question mark (?) is printed. This is not an error -- (you are simply creating and editing a new file. The text in the new file being worked on is kept in a special buffer file. Think of the buffer as a work space that you are going to be editing. You tell **ed** what to do to your text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lowercase. Each command is typed on a separate line and terminated with a <RETURN>. **Ed** makes no response to most commands -- there is no prompting or typing of messages. If at any time you make an error in the commands you type to **ed**, it will tell you by typing:

?

Adding Text to a File

To enter lines of text into the buffer, just type an "a" to append lines to the file, followed by a <RETURN>. Next, enter the lines of text you want, like this:

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

To stop appending, type a line that contains only a period on a line by itself. A period (.) is used to tell **ed** that you have finished appending. (You can also use <CONTROL-D>.)

Writing Out The Editing Buffer

It's likely that you will want to save your text for later use. To write out the contents of the **ed** editing buffer into a file, use the "write command, **w**, followed by the name of the file that you want to write to. This copies the buffer's contents to the specified file, destroying any

previous information in the file. For example, to save the text in a file named text, type:

```
w text
```

Leave a space between **w** and the filename. Writing a file just makes a copy of the text -- the buffer's contents are not disturbed, so you can go on adding lines to it. By default, **ed** writes out to the file that you named on the **ed** invocation line. Ed at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command.

Exiting The Editor

To terminate a session with **ed**, save the text you're working on by writing it to a file using the **w** command, and then type:

```
q
```

The system responds with the XENIX prompt character. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. Actually, ed will print "?", if you try to quit without writing. At that point, write out the text if you want; if not, type another **q**

A good way to operate is this:

```
ed file
[editing session]
w
q
```

This way, you can type **w** from time to time and be secure in the knowledge that if you got the filename right in the beginning, you are writing out to the proper file each time.

Printing Editing Buffer Contents

To print the contents of the buffer (or parts of it) on the terminal screen, use the "print" command, p. To do this, specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter "p". Thus, to print the first two lines of the buffer, (that is, lines 1 through 2) type:

```
1,2p
```

Suppose you want to print all the lines in the buffer. You'll then want to use the shorthand symbol for "the line number of the last line in the buffer" -- the dollar sign ($). Use it this way:

```
1,$p
```

This will print all the lines in the buffer (from line 1 to the last line).

To print the last line of the buffer, type:

```
$p
```

You can print any single line by typing the line number, followed by a p. Thus

```
1p
```

prints the first line of the buffer.

In fact, you can print any single line by typing just the line number; there's no need to type the letter p. So if you type

```
$
```

ed prints the last line of the buffer.

An area in which you can save typing effort in specifying lines is to use plus and minus as line numbers by themselves. For example

```
-
```

by itself is a command to move back up one line in the file.

Ed maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. The line most recently acted on is referred to with a period (.) and is called "dot." Dot is a line number in the same way that dollar ($) is; it means "the current line," or loosely, "the line you most recently did something to." You can find out the value of dot at any time by typing:

```
.=
```

Deleting Text

To delete text from the ed editing buffer, use the "delete" command, **d**. The lines to be deleted are specified for d exactly as they are for p:

starting-line,ending-lined

Thus, the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left in our example, as you can check by typing:

```
1,$p
```

Notice that $ now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Substituting Text

The "substitute" command, s, is used to replace a string of characters existing in the editing buffer with another string that you specify. This is the command that is used to change individual words or letters within a line or group of lines. For example, suppose that, due to a typing error, line 1 says:

```
Now is th time
```

The letter "e" has been left off of the word "the". You can use s to fix this up as follows:

```
1s/th/the/
```

This says to substitute for the characters "th", the characters "the", in line 1. To verify that the substitution has worked, type

```
p
```

to get

```
Now is the time
```

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

The syntax for the substitute command follows:

starting-line,ending-lines/pattern/replacement/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between starting-line and ending-line. Only the first occurrence on each line is changed, however. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution takes place, dot is not changed. This causes printing of a question mark (?) as a warning.

Thus, you can type

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text.

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

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it, to make sure it worked out right. (Notice that there is a **p** on the same line as the s command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.) To change all occurrences on the current line, you should type:

```
s/something/something else/g
```

where **g** stands for a global substitution of all occurrences on the line.

It's also legal to type

```
s/string//
```

which means "change the first string of characters to nothing," or, in other words, remove them.

Searching

Suppose you have the following three lines of text in the editing buffer:

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

Now, suppose you want to find the line that contains the word "their", so that you can change it to the word "the.". With only three lines in the buffer, it's pretty easy to keep track of which line the word "their" is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some textual pattern contained on the line.

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

```
/string of characters we want to find/
```

For example, the ed command

```
/their/
```

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

```
to come to the aid of their party.
```

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

```
?
```

Otherwise, ed prints the line it found.

X.8 Directory Manipulation and Travel

Printing Your Working Directory

All commands are executed relative to a "working" directory. The name of this directory is given by the **pwd** command, which stands for "print working directory." For instance, if your current working directory is /usr/joe, then when you type

```
pwd
```

you will get the output:

```
/usr/joe
```

You should always think of yourself as residing "in" your working directory.

Listing Directory Contents

The most basic directory command is **ls**. The **ls** command sorts and lists the names of the files and directories that reside in a given directory. By default, the contents of your working directory are listed. If arguments are given, then for each directory argument **ls** lists the contents of the given directory; for each file argument, **ls** repeats its name. For instance, if you type

```
ls
```

the output from the command might typically look like this:

```
dir1
dir2
dir3
file1
file2
file3
```

Using the same directory, the command

```
ls d*
```

would list the files within each of the directories dir1, dir2, and dir3.

When working at a terminal, it is sometimes distracting to see the list of names from the **ls** command scroll off

the screen. This problem can be avoided by using the lc command, which stands for "list in columns." Because names are printed in columns, more information can fit on the screen than with ls. A sample listing follows:

```
atfile      help        oem         size        v0
bin         lib         papers      src         v1
calendar    maketape    po          termcap     v2
cmds        memos       port        termnames   v5
convert     mgr         probs       test.s
doem        mkfs        rand        testdir
errs        msg         rand.c      ttc
errs.sh     nroff       sco         typeset
```

Note that when lc sends output through a pipe, no columns are sent. This is so that you can use lc like ls whenever you want. If you really want the output to a pipe to be columnar, you can force it with the -C switch. Thus, to send a columnar listing to the line printer, you would type:

```
lc -C | lpr
```

Note that lc also lets you recursively list a directory and all of its subdirectories by typing

```
lc -R
```

where the -R stands for recursive.

A command very similar to ls and lc is the l command. L gives an expanded "long" listing of a directory, producing an output that might look something like this:

```
total 501
drwxr-x--- 2 boris     272 Apr  5 14:33 dir1
drwxr-x--- 2 enid      272 Apr  5 14:33 dir2
drwxr-x--- 2 iris      592 Apr  6 11:12 dir3
-rw-r----- 1 olaf      282 Apr  7 15:11 file1
-rw-r----- 1 olaf       72 Apr  7 13:50 file2
-rw-r----- 1 olaf     1403 Apr  1 13:22 file3
```

Reading from left to right, the information given for each file or directory includes:

1. Permissions

2. Number of links

3. Owner

4. Size in bytes

5. Time of last modification

6. Filename

The information in this listing and how to change permissions are discussed below in Section X.9, "File and Directory Permissions."

Changing Your Working Directory

Your working directory represents your location in the file system: it is "where you are" in XENIX. To alter your location in the XENIX file system, you need only type:

```
cd
```

This changes your working directory to that of your home directory. To move to any given directory, simply specify that directory as an argument to cd. For instance

```
cd /usr
```

moves you to the /usr directory. Because you are always "in" your working directory, changing working directories is much like "traveling" from directory to directory.

Creating a Directory

To create a subdirectory in your working directory, use the mkdir command. For instance, to create a new directory named phonenumbers, simply type:

```
mkdir phonenumbers
```

After this command has been executed, a new but empty directory will exist in the your home directory.

Removing a Directory

To remove a directory located in your working directory, use the rmdir command. For instance, to remove the directory named phonenumbers from the current directory, simply type:

```
rmdir phonenumbers
```

Note that the directory phonenumbers must be empty before it can be removed; this prevents catastrophic deletions of files and directories.

Renaming a Directory

To rename a directory, use the **mv** command. Note that directories and their contents cannot be moved with the **mv** command; they can only be renamed. For instance, to rename the directory little.dir to big.dir, type:

```
mv little.dir big.dir
```

This is a simple renaming operation; no files are moved.

X.9 File and Directory Permissions

To determine the permissions associated with a given file or directory, use the **l** command. Permissions are given by the first ten characters of the output from this command. A sample output follows:

```
total 501
drwxr-x--- 2 boris     272 Apr  5 14:33 dir1
drwxr-x--- 2 enid      272 Apr  5 14:33 dir2
drwxr-x--- 2 iris      592 Apr  6 11:12 dir3
-rw-r----- 1 olaf      282 Apr  7 15:11 file1
-rw-r----- 1 olaf       72 Apr  7 13:50 file2
-rw-r----- 1 olaf     1403 Apr  1 13:22 file3
```

The first character is:

d	if the entry is a directory
-	if the entry is a ordinary file

The next nine characters are interpreted as three sets of three permissions each. The first set refers to user (owner) permissions; the next to permissions for others in a group associated with the file or directory; and the last to permissions for all other users. Within each set, the three characters indicate permission to read, to write, and to execute the file as a command, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for any included files or directories. In summary, these permissions are indicated as follows:

r if the file is readable

w if the file is writable

x if the file is executable

\- if the indicated permission is not granted

Changing Read, Write, and Execute Permissions

To change a file or directory's read, write, and execute permissions, you need only use one command: **chmod.** Several examples are included here to illustrate the syntax for this command. Remember that there are three distinct classes of users: user, group, and other, and that for each class all three permissions are either set or not set.

For example, presume that a file named filel exists with the following permissions:

```
-rw-r-----
```

To give filel read permission for all classes of users, type:

```
chmod a+r filel
```

Here, "a" stands for "all," which is a synonym for "ugo" where "u" stands for user, "g" for group, and "o" for other users. If no user class is specified, then "a" is normally the default, although this default may vary. The resulting permissions are:

```
-rw-r--r--
```

To give write and execute status to members of a group only, type:

```
chmod g+wx filel
```

For a filel with the attributes

```
-rw-------
```

the above command would alter attributes so that they looked like this:

```
-rw--wx---
```

To remove write and execute permission by the user

(owner) and group associated with filel, type:

```
chmod ug-wx filel
```

Note how letters are grouped to set specific user/permission combinations.

Changing Directory Search Permissions

Obviously, directories cannot be executable files, yet they, also, have an execute attribute that can be set. This attribute signifies search permission; if this permission is denied to a particular user, then that user cannot even list the names of the files in the directory.

For example, assume that the directory dirl has the following attributes:

```
drwxr-xr-x
```

To remove search and read permission for other users to examine dirl, type:

```
chmod o-rx dirl
```

The new attributes for dirl are then:

```
drwxr-x---
```

X.10 Information Processing

Sorting A File

One of the most generally useful file processing commands is **sort**. By default, **sort** sorts the lines of a file according to the ASCII collating sequence. For example, to sort a file named phonelist, type:

```
sort phonelist
```

In the above case, the sorted contents of the file are displayed on the screen. To create a sorted version of phonelist named phonesort, type:

```
sort phonelist >phonesort
```

Note that sort is useful in sorting the output from other commands. For example, to sort the output from execution of a who command, type:

```
who | sort >whosort
```

A wide variety of other sorting options are available for **sort.**

Searching for a Pattern in a File

The **grep** command selects and extracts lines from a file, printing only those lines that match a given pattern. A useful way to think of the name of this command is as a synonym for the word "grab."

For example, to print out all lines in a file containing the word "tty38", type:

```
grep 'tty38' file
```

In general, you should always enclose the pattern you are searching for in single quotes ('), so that special metacharacters are not expanded unexpectedly by the shell command interpreter.

X.11 Line Printers

Sending a File to the Line Printer

One of the most common operations that you will want to perform is printing files on the line printer. The most straightforward method for doing this is to type

```
lpr file1
```

for one file, or

```
lpr file1 file2 file3
```

for multiple files. Other common uses of **lpr** involve pipes. For instance, to paginate and print a file of raw text, type:

```
pr textfile | lpr
```

The pr and lpr commands are very often used together. As another example, to sort, paginate, and print a file, type:

```
sort datafile | pr | lpr
```

Getting Line Printer Queue Information

More than one file may be waiting to be printed at the line printer: XENIX does not require that the file be printed before the **lpr** command finishes. Instead, **lpr** makes sure only that the file is placed in a special directory where it will wait its turn to be printed. Thus, to examine the files in the line printer queue, type:

```
ls -l /usr/spool/lpd
```

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

01/83 SP

Printed in U.S.A.

TRS-XENIX System Fundamentals

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

TRS-XENIX SYSTEM

FUNDAMENTALS

ACKNOWLEDGEMENTS

This manual builds on the writing of many others. In many cases, the content here is identical, in whole or in part, to papers and manuals written at Bell Laboratories. In particular, Chapters 2 and 5 are adapted from papers written by Brian Kernighan. Chapter 6 and the glossary are adapted from documents written by Bill Joy and Mark Horton, then at the University of California at Berkeley. Chapter 7 is adapted from documents written by G. A. Snyder, J. R. Mashey, and S. R. Bourne. Chapter 8 is adapted from a paper written by Lee E. McMahon; Chapter 9 from a paper written by Robert Morris and Lorinda Cherry. The work of those mentioned above, and countless others, is gratefully acknowledged.

CONTENTS

CHAPTER 1

INTRODUCTION

CONTENTS

CHAPTER 1

1.1 Overview

This manual introduces the basic XENIX* system to the user with little or no XENIX experience, explaining the fundamental concepts and software needed to make effective use of the system. The XENIX system is Microsoft's improved and enhanced version of the popular UNIX* system. The XENIX system consists of a general-purpose multi-user, operating system and over one hundred utilities and application programs. In addition to the basic XENIX package described in this manual, two other XENIX packages are also available from Microsoft: the XENIX Software Development Package and the XENIX Text Processing Package. Other Microsoft software is also available.

Among the outstanding characteristics of the XENIX system are the following:

- A hierarchical, tree-structured file system that permits organization of files and directories in a logical and flexible way.
- A powerful, easy-to-use command language. Unlike other interactive command languages, the XENIX shell is a full programming language. The shell provides conditional, recursive, and iterative control structures, program variables, and a user environment that can be tailored to both individual and group needs.
- Simple and consistent naming conventions. Names can be absolute or relative to any directory in the file system.
- Device independent input and output: each physical device, from interactive terminals to main memory, is treated like a file, allowing uniform file and device input and output.
- A set of related text editors including a screen editor, a screen editor, and a steam editor.
- Flexible text processing facilities. In XENIX, commands exist to find and extract patterns of text from files, to compare and find differences between files, and to search through and compare directories. Also available, in the XENIX Text Processing Package, are text formatting, typesetting, and spelling error-detection facilities, as well as a facility for

---

* XENIX is a Trademark of the Microsoft Corporation.
UNIX is a Trademark of Bell Telephone Laboratories, Inc.

formatting and typesetting complex tables and equations.

- A sophisticated "desk-calculator" package.
- Mountable and dismountable file systems and volumes.
- A complete set of flexible directory and file protection modes that allow all combinations of read, write, and execute access for the owner of each file or directory, as well as for groups of users. Protection modes can be set dynamically.
- Facilities for creating, accessing, moving, and processing files, directories, or sets of these in a simple and uniform way.

1.2 Documentation Roadmap

This manual is organized as follows:

Chapter 1: Introduction
The chapter you are now reading gives an overview of the XENIX system and discusses the notational conventions used throughout the manual.

Chapter 2: Demonstration Run
This chapter gives you some hands-on experience in using the XENIX system.

Chapter 3: Basic Concepts
This chapter explains the fundamental concepts needed to understand how to use the system. Included here are sections on the file system, naming conventions, commands, and input and output.

Chapter 4: Frequently Used Procedures
This chapter explains how to perform everyday procedures using appropriate XENIX commands. This section separates these frequently used commands from the much larger set of all commands.

Chapter 5: Ed
This chapter describes how to use the line-oriented editor, ed.

Chapter 6: Vi
This chapter shows how to use the vi screen editor. Vi is the most sophisticated interactive editor available on XENIX.

Chapter 7: The Shell
This chapter describes the shell command language and how to write procedures that can be executed by the shell interpreter.

Chapter 8: Sed
This chapter describes the non-interactive "stream" editor named **sed**, similar in many ways to the XENIX editors **ed** and **vi**.

Chapter 9: BC
This chapter describes the use of BC, a sophisticated calculator program.

Appendix A: Glossary
This appendix contains a glossary of commonly used XENIX terms.

This manual makes no attempt to give information about installing, managing, and maintaining the system, nor does it discuss document preparation, software development, or many of the specialized utilities available in other XENIX packages. These subjects are dealt with in the following manuals:

The XENIX Operations Guide
This is a guide to installing, managing, and maintaining the whole system.

The XENIX Reference Manual
This is a comprehensive reference volume. A concise but complete description of each command is available here.

The XENIX Software Development Manual
This manual explains how to write programs that interface to the XENIX operating system and how to use some of the powerful tools available in the XENIX programming environment. This manual comes as part of the optional XENIX Software Development Package.

The XENIX Text Processing Manual
This manual explains how to use text processing and text formatting tools. This manual and comes as part of the optional XENIX Text Processing Package.

1.3 Notational Conventions

Throughout this manual, the following notational conventions are used:

boldface — Command names are given in boldface in the text of this manual; no boldface occurs in displays, except in syntax specifications for literal text. For example, **ls**, **date**, and **cd** are all the names of commands that you might type at the keyboard, and therefore all are in bold. An exception to this rule occurs for long chapters about a single command. In this case, the command name is made less conspicuous by either underlining or capitalization.

underlining — All filenames and pathnames are underlined. For example, text.file is a filename and /usr/mary is a pathname. Most command arguments are underlined as well, although in some cases these are in boldface. Words and phrases also may be underlined for emphasis. References to entries in the XENIX Reference Manual are underlined and include a section number in parentheses. For example, ls(1) refers to the entry for the **ls** command in Section 1, "Commands".

[brackets] — Brackets enclose optional arguments in syntax specifications.

<angle-brackets> — Angle brackets enclose the names of control characters and special function keys. Examples are <CONTROL-D>, <CONTROL-S>, <RETURN>, <INTERRUPT>, and <BKSP>.

ellipses... — Ellipses are used to indicate one or more entries of an argument in a syntax specification. For example, in the following syntax for the **mail** command, the ellipses indicate that one or more persons can be sent mail:

mail person ...

quotation marks — Quotation marks are used to set off multiple keystroke input. For example,

"ls -la ; date" is an example of a command line appearing in the body of the text.

Common abbreviations for ASCII characters are listed below:

<ESC>	Escape, Control-[
<RETURN>	Carriage return, Control-M
<LF>	Newline, Linefeed, Control-J
<BKSP>	Backspace, Control-H
<TAB>	Tab, Control-I
<BELL>	Bell, Control-G
<FF>	Formfeed, Control-L
<SPACE>	Space, 20 hexadecimal
<DEL>	Delete, 7F hexadecimal

1.4 Model-16 Console Keyboard

This section describes the mapping of Model-16 console keys to important XENIX characters not normally available on the Model-16 console keyboard.

XENIX Character	Model-16 Console
<INTERRUPT>	<BREAK>
<INTERRUPT>	<CONTROL-C>
<CONTROL-S>	<HOLD>
\ (backslash)	<CONTROL-/>
` (back quote)	<CONTROL-'>
| (vertical bar)	<CONTROL-!>
	<CONTROL-1>
<CONTROL-@> (null)	<CONTROL-2>
<CONTROL-]> (ASCII GS)	<CONTROL-3>
<CONTROL-^> (ASCII RS)	<CONTROL-4>
<CONTROL-_>	<CONTROL-5>
~ (tilde)	<CONTROL-6>
<CONTROL-\> (QUIT)	<CONTROL-7>
<DEL>	<CONTROL-_>

Note that the console's arrow keys are configured so that they send the following sequences to XENIX rather than their normal control sequences:

up arrow	<ESC>A
down arrow	<ESC>B
right arrow	<ESC>C
left arrow	<ESC>D

CHAPTER 2

DEMONSTRATION RUN

CONTENTS

2.1 Introduction

The purpose of this demonstration run is to help you get used to the main ideas of the XENIX system, so that you can quickly start to make effective use of it. It shows how to log in, how to type at your keyboard, what to do about mistakes in typing, how to enter commands, and how to log out.

2.2 Before You Log In

Before you can log into the system, your name must be added to the XENIX list of known users. You may have to add your name yourself, or someone else may be charged with this task; it all depends on the environment in which your system is used. In any case, see the XENIX Operations Guide for more information on adding users.

When you are given an account on the XENIX system you will also receive a user name, a password, and a login directory. Once you have these, all you need is a terminal on which you can log into the system. XENIX supports most terminals and you should have little problem in getting your terminal to work with XENIX. Once again, see the XENIX Operations Guide for more information on how to configure your terminal.

2.3 Logging In

To log in to XENIX, you must first have an up-and-running XENIX system. Normally, the system is sitting idle, with a simple "login:" prompt on the terminal screen. If the system displays nonsense characters, your terminal may be receiving information at the wrong speed and you should check your terminal's switches. If that fails, push the <BREAK> or <INTERRUPT> key a few times, slowly.

When you finally get a "login:" message, type your login name. Follow it by typing <RETURN>; the system will not do anything until you type it. If a password is required, you will be asked for it. The password that you type does not appear on the screen. This prevent others from viewing it. Don't forget to type <RETURN> after you type your password.

A successful login will produce a "prompt character," a single character that indicates the system is ready to accept commands. The prompt is usually a dollar sign ($) or a percent sign (%). You may also get a login message just before the prompt, or a notification that you have received mail sent by other users of the system.

2.4 Typing Commands

Once the prompt character appears, the system is ready to respond to commands typed at the terminal. Try typing

```
date
```

followed by <RETURN>. The system responds by displaying something like:

```
Mon Jun 16 14:17:10 EST 1983
```

Don't forget the <RETURN> after the command, or nothing will happen. <RETURN> won't be mentioned again, but don't forget it -- it has to be entered at the end of each command line. On some terminals <RETURN> may be labeled "ENTER" or "CR", but in all cases, the key performs the same function.

Another command you might try is who, which lists the names of everyone who is logged in to XENIX. A typical display from the who command might look something like this:

```
you     tty00   Jan 16    14:00
joe     tty01   Jan 16    09:11
ann     tty05   Jan 16    09:33
bob     tty11   Jan 16    13:07
```

The time given indicates when the user logged in; ttynn is the system name for each user's terminal, where nn is a unique two digit number.

If you make a mistake typing the command name, you will be told. For example, if you type

```
whom
```

you are told:

```
whom: not found
```

Note that case is significant in XENIX -- the commands

```
who
```

and

```
WHO
```

are not the same; this differs from some operating systems, where case doesn't matter.

Now let's try using the **echo** command. Type:

```
echo hello world!
```

Echo simply does what its name says and echos the rest of the command line on your terminal:

```
hello world!
```

Now try this:

```
echo hello world! >greeting.file
```

This time the **echo** command sends its output to a file, instead of to your terminal. Note the use of the greater than sign (>) to "redirect" the output of the command. (See Section 3.5.1 for more information on redirection.) Now type

```
ls
```

to list the name of the file, greeting.file. To look at the contents of greeting.file display it by typing:

```
cat greeting.file
```

("Cat" stands for concatenate.) The output to your terminal is the echoed message:

```
hello world!
```

2.5 Strange Terminal Behavior

Occasionally, your terminal may act strangely. You can often fix this behavior by logging out and logging back in: this will reset your terminal characteristics. If logging out and back in doesn't work, read the description of the command stty(1) in the XENIX Reference Manual for more information about setting terminal characteristics.

Some terminals automatically expand tabs into an appropriate number of spaces; others do not. To get tabs expanded into spaces for terminals without automatic tab expansion, type:

```
stty -tabs
```

The system will then convert each tab into the correct number of spaces for you. If your terminal does have computer-settable tabs, the command **tabs** correctly sets the tab stops for you.

2.6 Mistakes in Typing

If you make a mistake in typing while entering a command line, there are two ways to edit the line, provided you have not yet typed <RETURN>. Pressing the <BKSP> key causes the last character typed to be erased. In fact, backspacing with the <BKSP> key erases characters back to the beginning of the line, but not beyond. Thus, if you type badly, you can correct as you go. For example, typing

```
dd<BKSP>atte<BKSP><BKSP>e<RETURN>
```

is the same as typing just

```
date<RETURN>
```

The XENIX kill character, <CONTROL-U>, erases all of the characters typed so far on the current input line. So, if the line is irretrievably fouled up, type <CONTROL-U> and start the line over.

What if you must enter a <BKSP> or <CONTROL-U> as part of the text? If you must, precede it with a backslash (\), so that the character loses its special, erase meaning. To enter a <BKSP> or <CONTROL-U> in text, type "\<BKSP>" or "\<CONTROL-U>". The system always prints a new line on your terminal after your <CONTROL-U>, even if preceded by a backslash. Don't worry -- the <CONTROL-U> will have been recorded.

To erase a backslash, backspace twice with the <BKSP> key, as in "\<BKSP><BKSP>". The backslash is used extensively in XENIX to indicate that the following character is in some way special. Note that the functions performed by <BKSP> and <CONTROL-U> are available on all XENIX systems; however, the keys used to perform these functions can be changed and set by the user.

2.7 Read-Ahead

XENIX has full read-ahead, which means that you can type as fast as you want, whenever you want, and XENIX will remember what you have typed. If you enter any text while a command is displaying text on the screen, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. Therefore, you can type several commands (i.e., "type ahead") one after another without waiting for the first to finish. Note that the one place that this doesn't work is when you login; type ahead does not work until after you

have entered your password and you see the dollar sign ($) prompt for command input.

2.8 Stopping a Program

You can abort the execution of most programs and commands by pressing the <INTERRUPT> key (perhaps called <DEL>, <DELETE>, <CONTROL-C>, or <RUBOUT> on your terminal). The <BREAK> key found on many terminals can also be used. Inside some programs, like the text editor, <INTERRUPT> stops whatever the program is doing without aborting the program itself. Throughout this manual when we say "send an interrupt" we mean press the <INTERRUPT> key.

2.9 Logging Out

To end a session with XENIX, you must log out. This is done by typing a <CONTROL-D>. It is not sufficient just to turn off the terminal, since this does not log you out.

2.10 Further Learning

For further learning, you will want to use the on-line help facility: **learn**. Simply type

```
learn
```

and it will give you instructions. **Learn** is especially useful when first learning how to use the system, so you may want to investigate it now.

This ends the demonstration run. What you have learned so far is enough to keep you out of harm's way. You will want to learn how to use the XENIX shell and a text editor. The shell language is the language with which you speak to the system and execute commands. Text editors are used to create and edit files, documents, and programs. The available editors and the shell are discussed thoroughly in Chapters 5, 6, and 7.

The next two chapters introduce you to the concepts and commands that you will need to know each time you log in to XENIX. Be sure to read them before you try to do anything you think difficult or dangerous. It is especially important to understand how the directory system is organized and how files are named.

CHAPTER 3

BASIC CONCEPTS

CONTENTS

3.1 Introduction

This chapter gives you an understanding of the basic concepts that you need to function in the XENIX environment. It discusses the XENIX file system, naming conventions, commands, and input and output. After reading this chapter you will have an understanding of how the system's files, directories, and devices are organized and named, how commands are entered, and how a command's input and output can be manipulated.

3.2 File System

A file system is a set of files organized in a logical fashion. In XENIX, this set of files consists of all available system resources including data files, programs, line printers, and disks. Thus, the XENIX file system is more generally a system for accessing all of the resources of the system.

To logically structure the files and resources of the system, the XENIX file system is organized hierarchically into a "tree-structure." See Figure 3-1 for an illustration of a typical tree-structured file system. In this tree of files, the root of the tree is at the top and branches of the tree grow downward. Directories correspond to nodes in the tree; ordinary files correspond to "leaves." If a directory contains a downward branch to other files or directories, then those files and directories are "contained" in the given directory. It is possible to specify any file in the system by starting at the root (where the root is at the top) and traveling down any of the branches to the desired file. Similarly, you can specify any file in the system, relative to any directory. Specification of these files depends on a knowledge of XENIX naming conventions, discussed in Section 3.3.

Privacy and security for files and directories can be arranged. Each file and directory has read-write-execute permissions that can be set to control access by the owner, by a group of users, and by everyone else.

Basic Concepts

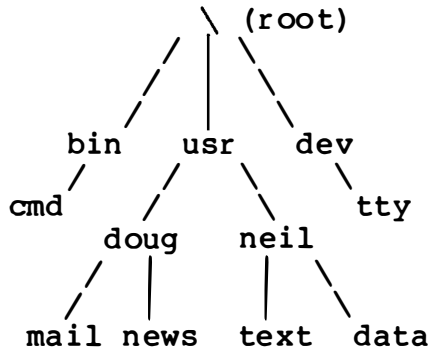


Figure 3-1. A Typical File System

In the typical tree-structured file system of Figure 3-1, the "tree" grows downward. The names bin, usr, dev, doug, and neil all represent directories, and are all nodes in the tree. In XENIX the name of the root directory is given the one character name, "/". The names mail, news, text, and data all represent normal data files, and are all "leaves" of the tree. Note that cmd, which here is an executable command, is also a leaf and therefore a file. The name tty represents a terminal and is also represented in the tree.

3.2.1 Files

The file is the fundamental unit of the file system. Conceptually, everything is treated as a file. However, there are really three different type of files: ordinary files (what we usually mean when we say "file"), special files, and directories. Each of these file types is discussed below:

Ordinary Files
: Ordinary files typically contain textual information such as documents, data, or program sources. Executable binary files are also of this type.

Special Files
: A special file is one that corresponds to a physical device of some sort, such as a disk, a line printer, a terminal, or system memory. To the XENIX user, special files can be treated like ordinary files. However, the internal handling of a printer or terminal is much different from that of an ordinary disk file, and the operations that can be performed on devices vary from those that can be performed on ordinary files.

Directories

Directories are read-only files containing information about the files or directories that are conceptually (but not physically) contained within them. The nesting of directories in other directories is the way in which XENIX implements its characteristic tree-structured directory system. Directories are discussed further in the next section.

3.2.2 Directory Structure

With multiple users and multiple projects, the number of files in a file system can proliferate rapidly. Fortunately, as explained earlier, XENIX organizes all files into a tree-structured directory hierarchy. Each user of the system has his own personal directory. Within that directory, the user may have directories or other subdirectories owned and controlled only by the user.

When you log in to XENIX, you are "in" your directory. Unless you take special action when you create a file, the new file is created in your working directory. This file is unrelated to any other file of the same name in someone else's directory.

A diagram of part of a typical user directory is shown in Figure 3-2.

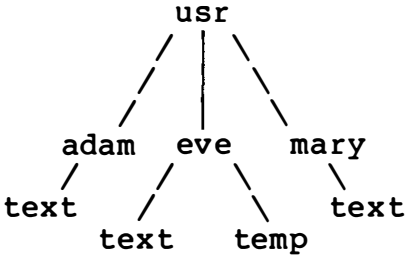


Figure 3-2. A Typical User Directory

In Figure 3-2, the usr directory contains each user's own personal directory. Notice that Mary's file named text is unrelated to Eve's. This is not important if all the files of interest are in Eve's directory, but if Eve and Mary work together, or if they work on separate but related projects, this division of files becomes handy indeed. For example, Mary could print Eve's text by typing:

```
pr /usr/eve/text
```

Similarly, Eve could find out what files Mary has by typing:

```
ls /usr/mary/*
```

3.3 Naming Conventions

Now that we have discussed what the XENIX file system is, and what it consists of, we need to use filenames in a more precise way. The first thing to remember is that every single file, directory, and device in XENIX has both a filename and an absolute pathname. The absolute pathname is unique to all names in the system; filenames are unique only within directories and need not be unique system-wide. This is similar to someone who's "absolute" name is John Robert Smith, but whom everyone calls John. The name John need not be unique, although it will greatly simplify life if John Robert Smith *is* a unique name.

3.3.1 Filenames

A simple filename is a sequence of 1-14 characters other than a slash (/). Every single file, directory, and device in the system has a filename. Filenames are used to uniquely identify directory contents. Thus, no two names in a directory may be the same. However, filenames in different directories may be identical.

3.3.2 Pathnames

A pathname is a sequence of directory names followed by a simple filename, each separated from the previous one by a slash. If a pathname begins with a slash, the search for the file begins at the *root* of the entire tree. Otherwise, it begins at the user's *current directory* (also known as the *working directory*). A pathname beginning with a slash is called a *full* (or *absolute*) *pathname* because it does not vary with regard to the user's current directory. A pathname *not* beginning with a slash is often called a *relative pathname*, because it specifies a path relative to the current directory. The user may change the current directory at any time by using the **cd** command.

In most cases, a filename and its corresponding pathname may be used interchangeably.

3.3.3 Sample Names

Some sample names follow:

/	The absolute pathname of the root directory of the entire file system.
/bin	The directory containing most of the frequently used XENIX commands.
/usr	The directory containing each user's personal directory. The subdirectory, /usr/bin contains frequently used XENIX commands not in /bin.
/dev	The directory containing files corresponding to each available physical device (e.g., terminals, line printers, and disks).
/lib	The directory containing special data files used by some standard commands.
/tmp	This directory contains temporary scratch files.
/usr/joe/project/A	This is a typical full pathname. This one happens to be a file named A in the directory named project belonging to the user named joe.
bin/x	A relative pathname; it names the file x in subdirectory bin of the current working directory. If the current directory is /, it names /bin/x. If the current directory is /usr/joe, it names /usr/joe/bin/x.
file1	Name of an ordinary file in the current directory.

Each user resides "in" a directory called the current directory. All files and directories have a "parent" directory. This directory is the one immediately above and "containing" the given file or directory. The XENIX file system provides special shorthand notations for this directory and for the current directory:

- . The shorthand name of the current directory. Thus ./filexxx names the same file as filexxx, if such a file exists in the current directory.

.. The shorthand name of the current directory's parent directory. If you type

```
cd ..
```

then the parent directory of your current working directory becomes your new current directory.

Although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. For instance, the dash (-) is used in specifying command options, and should be avoided when naming files. To avoid pitfalls, you will do well to use only letters, numbers, and the period.

3.3.4 Special Characters

XENIX provides a pattern-matching facility for specifying sets of filenames that match particular patterns. For example, examine the problem that occurs when naming the parts of a large document, such as a book. Logically, it can be divided into many small pieces: chapters or perhaps sections. Physically, it must be divided too, since the XENIX editor, ed, cannot handle really big files. Thus, you should construct a document as a number of files. For example, you might have a separate file for each chapter:

```
chap1
chap2
...
```

Or, if each chapter were broken into several files, you might have:

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

You could then tell at a glance where a particular file fits into the whole.

There are other advantages to a systematic naming convention that are not so obvious. What if you wanted to print the whole book on the lineprinter? You could type

```
lpr chap1.1 chap1.2 chap1.3 ...
```

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut: a sequence of similar names can be specified with the use of two special "wild card" characters.

For example, you can type:

```
lpr chap*
```

The asterisk (*), called "star" in XENIX, means "zero or more characters of any type," so this translates into "print all files whose names begin with the word "chap", listed in alphabetical order."

This shorthand notation is not a unique property of the lpr command; it is a system-wide service of the shell program that interprets commands, sh. Using this fact, you list the names of the files in the book by typing:

```
ls chap*
```

This produces

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

The star is not limited to the last position in a filename; it can be used anywhere and can occur several times. As a special case, a star by itself matches every filename, so

```
rm *
```

removes all your files.

The star is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4, and 9. Then you can say

```
lpr chap[12349]*
```

The brackets ([and]) mean "match any of the characters inside the brackets." A range of consecutive letters or digits can be abbreviated, so you can also do this with

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

(Note that this does not match forty-nine filenames, but

only five.) Letters can also be used within brackets: "[a-z]" matches any character in the range "a" through "z".

The question mark (?) matches any single character, so

```
ls ?
```

lists all files that have single-character names, and

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

lists information about the first file of each chapter (i.e., chap1.1, chap2.1, ...).

Of these pattern matching conventions, the star (*) is certainly the most useful, and you should get used to it at once.

If you should ever need to turn off the special meaning of any of the special characters (*, ?, and [...]) enclose the entire argument in single quotes. For example, the following command will print out only files named "?" rather than all one character filenames:

```
ls '?'
```

All of these special pattern matching features are further discussed in Chapter 7, The Shell.

3.4 Commands

Commands are used to invoke executable programs. When you type the name of a command, the XENIX shell looks for a program with the given name to execute. If the shell finds an executable program, it will execute it. Commands may also contain switches that specify options or other arguments as needed by the program. Commands are entered on a command line that is read by the shell. Command lines are discussed in the following subsection.

3.4.1 Command Line

Whether typing at the terminal, or executing commands from a file, XENIX always reads commands from command lines. The command line is a line of characters that is scanned and read by the shell command interpreter to determine what to do next. When you are typing at a terminal, you are editing a line of text called the command-line buffer that becomes a command line only when you type <RETURN>. This command-line

buffer can be edited with the <BKSP> and <CONTROL-U> keys. Typing <RETURN> causes the command-line buffer to be submitted to the shell as a command line. The shell reads the command line and executes the appropriate command. If you type <INTERRUPT> before you type <RETURN>, then the command-line buffer is aborted. Multiple commands can be entered on a single command line so long as they are separated by a semicolon (;). For example, the following prints out the current date and the name of the current working directory:

```
date ; pwd
```

Commands can be submitted for processing in the background by appending an ampersand to the command. Thus

```
mv file1 file2 file3 file4 dir1&
```

will move the files file1, file2, file3, and file4 to the directory dir1 without tying up your terminal. You can execute other commands from your terminal in the foreground while the mv command executes in the background.

3.4.2 Syntax

The general syntax for commands is as follows:

cmd [switches][arguments] [filenames]

In practically all cases, command names are all lowercase. Switches are flags that select various options available when executing the command. Switches are optional and always precede other arguments and filenames. Switches consist of a dash prefix (-) and an identifying alphanumeric character. Some switches are also prefixed by a plus sign (+). Switches can often be grouped as a single switch as in:

```
ls -arl
```

Here the -a switch (pronounced "minus a") selects the option which lists all files in the directory. The -r switch selects the option which causes the names in the directory to be sorted in reverse alphabetical order. And the -l switch selects the option which causes listing of a long format for each directory entry.

Sometimes switches must be given separately, as in:

```
copy -v -a source dest
```

Here the -v switch specifies a verbose option. The -a switch tells the **copy** command to ask the user before copying the two given directories.

Arguments of various types can also be given, such as search strings, as in:

```
grep 'string of text' outfile
```

In the above example, "string of text" is an argument and is the search string that the **grep** command searches for in the file outfile. Outfile, itself, is a filename argument that specifies the name of a file required by the command.

In most cases, commands are executable object files compiled from C programs. In some cases, commands are executable command files called "shell procedures." Shell procedures are discussed in Chapter 7, The Shell.

3.5 Input and Output

XENIX handles input and output from commands in a unique way: it assumes that input and output, by default, are associated with the terminal from which the command originates. That is, input comes from the keyboard and output goes to the terminal screen. To illustrate typical command input and output, type:

```
cat
```

This command now expects input from your keyboard. It will accept as many lines of text as you can type as input, until you type a <CONTROL-D> as an end-of-file indicator. For example, type:

```
this is two lines
of input
<CONTROL-D>
```

When you type the <CONTROL-D>, input ends and output begins. The **cat** command then immediately outputs the two lines that you typed -- since output is sent to the terminal screen by default, that is where the two lines are sent. Thus, the complete session will look like this on your terminal screen:

```
$ cat
this is two lines
of input
this is two lines
of input
$
```

The flow of command input and output can be "redirected" so that input comes from a file instead of from the terminal keyboard, and so that output goes to a file or to a line printer, instead of to the terminal screen. In addition, "pipes" can be created that allow the output from one command to become the input to another. Redirection and pipes are the subjects of the next two subsections.

3.5.1 Redirection

It is universal in XENIX systems that a file can replace the terminal for either input or output. For example

```
ls
```

displays a list of files on your terminal screen. But if you say

```
ls >filelist
```

a list of your files is placed in the file filelist (which is created if it does not exist). The output redirection symbol (>) means "put the output from the command into the following file, rather than display it on the terminal screen." As another example, you could combine several files into one by capturing the output of **cat** in a file:

```
cat fl f2 f3 >temp
```

The output append symbol (>>) operates very much like the output redirection symbol (>) does, except that it means "add to the end of." That is

```
cat filel file2 file3 >>temp
```

means to concatenate filel, file2, and file3 to the end of whatever is already in temp, instead of overwriting and destroying the existing contents. As with normal output redirection, if temp doesn't exist, it is created for you.

In a similar way, the input redirection symbol (<) means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a

script of editing commands and put them into a file called script. Then you could execute the commands in the script on a file using the XENIX editor by typing:

```
ed file <script
```

As another example, you could use ed to prepare a letter in file letter.txt, then send it to several people with

```
mail adam eve mary joe <letter.txt
```

3.5.2 Pipes

One of the major innovations of the XENIX system is the concept of a pipe. A pipe is simply a way to connect the output of one command to the input of another command, so that the two run as a sequence of commands called a pipeline.

For example

```
pr frank.id george.id hank.id
```

prints the files named frank.id, george.id, and hank.id, beginning each on a new page. Suppose you want them run together instead. You could type:

```
cat frank.id george.id hank.id >temp
pr <temp
rm temp
```

But this is more work than is necessary. What we want is to take the output of cat and connect it to the input of **pr**. So we use a pipe:

```
cat frank.id george.id hank.id | pr
```

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

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program wc counts the number of lines, words, and characters in its input, and who prints a list of currently logged on people, one per line. Thus,

```
who | wc
```

tells how many people are logged on. And of course

```
ls | wc
```

counts your files.

Any program that reads from the terminal keyboard can read from a pipe instead. Any program that displays output to the terminal screen can send input to a pipe. You can have as many elements in a pipeline as you wish.

Many XENIX programs are written so that they take their input from one or more files, if file arguments are given. If no arguments are given, they read from the terminal keyboard, and thus can be used in pipelines. For example

```
pr -3 albert.txt bernard.txt carl.txt
```

prints, in order, the files albert.txt, bernard.txt, and carl.txt. But in

```
cat albert.txt bernard.txt carl.txt | pr
```

pr prints the concatenation of these files coming down the pipeline. The difference is that here, albert.txt, bernard.txt, and carl.txt are run together and then treated as one file rather than three.

CHAPTER 4

FREQUENTLY USED PROCEDURES

CONTENTS

4.1 Introduction

This chapter explains how to perform frequently used procedures, where a procedure is some task that may require one or more steps to complete. The commands used to preform these procedures are discussed more thoroughly in the XENIX Reference Manual.

4.2 System Access

Logging In

The procedure for gaining access to the system is to respond to the "login:" prompt by typing your user name followed by a <RETURN>. Then respond to the password: prompt with your password. For example, a login for the user joe might look like this:

```
login:joe
password:abracadabra
```

Note that the password is not shown on the terminal screen.

Gaining Access to a Group

When a file or directory is created, it has group permission attributes associated with it that control access to the file by other members of the same group. This means that some files can only be accessed if you are a member of the right group. When you log in to XENIX, you are automatically associated with a default group.

To change groups, you must go through a procedure similar to that for logging in. As with the login procedure, a password may be required to switch your association from one group to another. (Note, however, that there is no XENIX command to set or change group passwords.)

Assume, for instance, that you are associated with the group named "engineers". To change your group affiliation to a new group named "managers" you would type:

```
newgrp managers
```

If the new group requires a password, you will be prompted for it. For example, if the password is "AbleBaker" then you would respond to the password prompt:

```
password:AbleBaker
```

Note that "AbleBaker" would not be shown on the terminal screen.

Logging Out

The logout procedure is simple -- all you need to do is type:

```
<CONTROL-D>
```

In general, <CONTROL-D> signifies the end-of-file in XENIX, and is often used within programs to signal the end of input from the keyboard. In such cases, <CONTROL-D> will not log you out, but simply terminate input to a particular program. This means, that it may sometimes be necessary to type <CONTROL-D> several times before you can log yourself out.

Changing Your Password

To change your password, use the **passwd** command. For the user joe, a session might go like this:

```
Changing password for joe
Old password: palooka
New password: Bazookah
Retype new password: Bazookah
```

To maintain security, none of your responses are shown on the screen. It is best to mix upper- and lowercase letters and to pick a password greater than five characters in length. These measures should be taken to foil automated attempts at guessing your password. Note that the system ignores characters in the password beyond the eighth character.

4.3 Terminal Configuration

Setting Terminal Options

There are a number of terminal options that can be set with the command **stty**. When entered without parameters, **stty** displays the current terminal settings. For example, typical output might look like this:

```
speed 9600 baud
erase '^h' ; kill '^u'
even -nl
```

Each of the above characteristics can be set with **stty**. For more information, see stty(1) in the XENIX Reference Manual.

Setting Tabs

The tab stops for a particular terminal can be specified with the **tabs** command. **Tabs** takes the name of a terminal as its only parameter. For example, to reset tab stops for your terminal, type:

```
tabs
```

That is all you have to do.

Changing Terminals

If you ever need to login to XENIX on a terminal of a type different than the terminal you normally use, you may need to change the shell TERM variable. This is normally set to the proper default terminal when you login, but if you switch terminals, you'll need to type something like:

```
TERM=termname; export TERM
```

where termname is the name of a known terminal. A wide variety of terminals are supported; terminal names are listed in the system file named /etc/termcap.

4.4 Process Control and Command Line Editing

Placing A Process in the Background

Normal commands executed at the keyboard are executed in strict sequence: one must finish executing before the next can begin. Executing commands of this type are called foreground processes. A background process, in contrast, need not finish executing before you execute your next command. Background commands are especially useful for commands that may take several minutes or even hours to complete, because they can be placed in the background while you continue executing other commands at your terminal.

To place a process in the background, type an ampersand (&) at the end of the command. For example, to print out several files, while simultaneously continuing with whatever else you have to do, type:

```
lpr file1 file2 file3&
```

Note that when processes are placed in the background, you lose control of them as they execute. For instance, typing <INTERRUPT> does not abort a background process. You must use the **kill** command instead, described below.

Killing a Process

To abort execution of a foreground process press your terminal's <INTERRUPT> key. This kills whatever foreground command you have running. To kill all of your processes that are executing in the background, type:

```
kill 0
```

To kill only a specified process executing in the background, first type:

```
ps
```

Ps displays the Process Identification Numbers (PIDs) of your existing processes:

```
 PID TTY TIME CMD
3459 03  0:15 -sh
4831 03  1:52 cc program.s
5185 03  0:00 sh -c ps
```

Next, you might type

```
kill 4831
```

where 4831 is the PID of the process that you want killed.

Warning: Killing a process associated with the editor **vi**, may leave the terminal in a strange mode. Also, temporary files that are normally created when a command starts and deleted when the command finishes, may be left behind after a **kill** command. Temporary files are normally kept in the directory /tmp.

Erasing A Command Line

When entering commands, typing errors will occur. To erase the current command line so that you can start retyping a new one, enter a <CONTROL-U>, as shown below

```
kat file2<CONTROL-U>
```

and then enter the correct command on the next line:

```
cat file1<RETURN>
```

In the above example, the first line is aborted and automatically a newline is generated so that typing may resume. You then enter the correct command line.

Halting Screen Output

In many cases, you will be examining the contents of a file on the terminal screen. For longer files, the contents will often scroll off the screen faster than you can examine them. To temporarily halt a program's output to the terminal screen, type <CONTROL-S>. To resume output, type a <CONTROL-Q>.

4.5 Status Information

Finding Out Who is on the System

The **who** command lists the names, terminal line numbers, and login times of all users currently logged on the system. For example, type:

```
who
```

This command should produce something like the following output on your terminal screen:

```
arnold   tty02   Apr  7 10:02
daphne   tty21   Apr  7 07:47
elliot   tty23   Apr  7 14:21
ellen    tty25   Apr  7 08:36
gus      tty26   Apr  7 09:55
adrian   tty28   Apr  7 14:21
```

The **finger** command also provides information about the users of the system. For more information, see finger(1) in the XENIX Reference Manual.

Finding out What Processes are Running

Because processes can be placed in the background for processing, it is not always obvious which processes you are responsible for. The ps command stands for "process status" and lists information about currently running processes associated with your terminal. For instance, the output from a ps command might look like this:

```
 PID TTY TIME CMD
10308 38  1:36 ed chap02.man
   49 38  0:29 -sh
11267 38  0:00 sh -c ps
```

The PID column gives a unique process identification number that can be used to kill a particular process. The TTY column gives the terminal that the process is associated with. The TIME column gives the cumulative execution time for the process.

Getting Line Printer Information

At times it may be necessary to know how many files are queued up at the line printer. This information can be found by listing the directory in which queued files reside. Typically, this is in /usr/spool/lpd. To examine the relevant files type:

```
ls -l /usr/spool/lpd/[df]*
```

4.6 File Manipulation

Creating a File

To create an empty file, simply type:

```
>filename
```

Here, filename is the name of the newly created file. The greater-than sign (>) is used to redirect output from the terminal to a file. In this special case no information is sent. In general, new files are created by commands as needed. To edit the contents of a file, you should use one of the editors ed or Vi. Ed is discussed in Chapter 5, Ed; Vi is discussed in Chapter 6, Vi.

Displaying File Contents

There are two programs available to display the contents of a file: cat and more. The cat command displays the contents of a file on the default standard output file which is the terminal screen. For example, the following command displays the contents of filel on the screen:

```
cat filel
```

Cat can also display the contents of more than one file as in

```
cat filel file2
```

The more command performs a similar function, except that it displays a file's contents a line or screen at a time. After invoking more, file contents can be examined a line at a time by typing <RETURN>. More tells you what percentage of the file has been viewed

on the bottom status line. If you type the letter "z" or a space, then the next screenful is displayed. A listing of other **more** commands is obtained by typing "h" for help.

Combining Files

Cat, as mentioned above, is normally used to send the contents of files to the terminal screen. However, the name **cat** comes from the word concatenate, and **cat** is also frequently used to combine files into some other new file. Thus, to combine the two files named filel and file2, and to create a new file named bigfile, type:

```
cat filel file2 >bigfile
```

Note here that we are putting the contents of the two files into a new file with the name bigfile. The greater than sign (>) is used to redirect output of the **cat** command to the new file.

Moving a File

With the **mv** command, two ways of moving a file are supported:

1. Moving a file so that it now has a new name. For instance, to move a file named text to a new file named book, type:

   ```
   mv text book
   ```

   After this move completes, no file named text will exist in the working directory.

2. Moving a file into a specified directory. In this case, you give the name of the destination directory as the final name in the move command. For instance, to move filel and file2 into the directory named /tmp, type:

   ```
   mv filel file2 /tmp
   ```

   The two files you have moved no longer exist in your working directory, but files with the same names now exist in the directory /tmp. The above command has exactly the same effect as typing the following two commands:

```
mv file1 /tmp/file1
mv file2 /tmp/file2
```

Remember that the **mv** command always checks to see if the last argument is the name of a directory, and, if so, all files designated by filename arguments are moved into that directory.

Renaming a File

To rename a file, you simply "move" the file to a file with the new name: the old name of the file is removed. Thus, to rename the file anon to johndoe, type:

```
mv  anon johndoe
```

Note that moving and renaming a file are essentially identical operations.

Copying a File

There are two forms of the **cp** command: one in which files are copied into a directory, and another in which a file is copied to another file. Thus, to copy three files into a directory named filedir, type:

```
cp file1 file2 file3 filedir
```

In the above command, three files are copied into the directory filedir; the original versions still reside in the working directory. There is a one-to-one correspondence between the names in the two directories.

To create two copies of a file in your own working directory, you must rename the new copy. To do this, the copy command can be invoked as follows:

```
cp file clone-of-file
```

After the above command has executed, two files with identical contents reside in the working directory. See also the **copy** command, for recursively copying directories.

Deleting A File

To delete or remove files, simply type:

```
rm file1 file2
```

In the above command, the files file1 and file2 are removed from your working directory. The command

```
rm *
```

removes all files from the working directory (subdirectories are not removed).

4.7 Directory Manipulation and Travel

Printing Your Working Directory

All commands are executed relative to a "working" directory. The name of this directory is given by the pwd command, which stands for "print working directory". For instance, if your current working directory is /usr/joe, then when you type

```
pwd
```

you will get the output:

```
/usr/joe
```

You should always think of yourself as residing "in" your working directory.

Listing Directory Contents

The most basic directory command is ls. The ls command sorts and lists the names of the files and directories that reside in a given directory. By default, the contents of your working directory are listed. If arguments are given, then for each directory argument ls lists the contents of the given directory; for each file argument, ls repeats its name. For instance, if you type

```
ls
```

the output from the command might typically look like this:

```
dir1
dir2
dir3
file1
file2
file3
```

Using the same directory, the command

```
ls d*
```

would list the files within each of the directories dir1, dir2, and dir3.

When working at a terminal, it is sometimes distracting to see the list of names from the ls command scroll off the screen. This problem can be avoided by using the lc command, which stands for "list in columns." Because names are printed in columns, more information can fit on the screen than with ls. A sample listing follows:

```
atfile      help        oem         size        v0
bin         lib         papers      src         v1
calendar    maketape    po          termcap     v2
cmds        memos       port        termnames   v5
convert     mgr         probs       test.s
doem        mkfs        rand        testdir
errs        msg         rand.c      ttc
errs.sh     nroff       sco         typeset
```

Note that when lc sends output through a pipe, no columns are sent. If you really want the output to a pipe to be columnar, you can force it with the -C switch. Thus, to send a columnar listing to the line printer, you would type:

```
lc -C | lpr
```

Note that lc also lets you recursively list a directory and all of its subdirectories by typing

```
lc -R
```

where the -R stands for recursive.

A command very similar to ls and lc is the l command. L gives an expanded "long" listing of a directory, producing an output that might look something like this:

```
total 501
drwxr-x--- 2 boris     272 Apr  5 14:33 dir1
drwxr-x--- 2 enid      272 Apr  5 14:33 dir2
drwxr-x--- 2 iris      592 Apr  6 11:12 dir3
-rw-r----- 1 olaf      282 Apr  7 15:11 file1
-rw-r----- 1 olaf       72 Apr  7 13:50 file2
-rw-r----- 1 olaf     1403 Apr  1 13:22 file3
```

Reading from left to right, the information given for each file or directory includes:

1. Permissions
2. Number of links
3. Owner
4. Size in bytes
5. Time of last modification
6. Filename

The information in this listing and how to change permissions are discussed below in Section 4.8, "File and Directory Permissions."

Changing Your Working Directory

Your working directory represents your location in the file system: it is "where you are" in XENIX. To alter your location in the XENIX file system, you need only type:

```
cd
```

This changes your working directory to that of your home directory. To move to any given directory, simply specify that directory as an argument to cd. For instance

```
cd /usr
```

moves you to the /usr directory. Because you are always "in" your working directory, changing working directories is much like "traveling" from directory to directory.

Creating a Directory

To create a subdirectory in your working directory, use the mkdir command. For instance, to create a new directory named phonenumbers, simply type:

```
mkdir phonenumbers
```

After this command has been executed, a new but empty directory will exist in the your home directory.

Removing a Directory

To remove a directory located in your working directory, use the rmdir command. For instance, to remove the directory named phonenumbers from the current directory, simply type:

```
rmdir phonenumbers
```

Note that the directory phonenumbers must be empty before it can be removed; this prevents catastrophic deletions of files and directories. If you want to live dangerously, it is possible to recursively remove the contents of a directory using the rm command, but that will not be explained here. See rm(1) in the XENIX Reference Manual for more information.

Renaming a Directory

To rename a directory, use the mv command. Note that directories and their contents cannot be moved with the mv command; they can only be renamed. For instance, to rename the directory little.dir to big.dir, type:

```
mv little.dir big.dir
```

This is a simple renaming operation; no files are moved.

4.8 File and Directory Permissions

To determine the permissions associated with a given file or directory, use the l command. Permissions are given by the first ten characters of the output from this command. A sample output follows:

```
total 501
drwxr-x--- 2 boris     272 Apr  5 14:33 dirl
drwxr-x--- 2 enid      272 Apr  5 14:33 dir2
drwxr-x--- 2 iris      592 Apr  6 11:12 dir3
-rw-r----- 1 olaf      282 Apr  7 15:11 filel
-rw-r----- 1 olaf       72 Apr  7 13:50 file2
-rw-r----- 1 olaf     1403 Apr  1 13:22 file3
```

The first character is:

d if the entry is a directory

\- if the entry is a ordinary file

The next nine characters are interpreted as three sets of three permissions each. The first set refers to user (owner) permissions; the next to permissions for others in a group associated with the file or directory; and the last to permissions for all other users. Within each set, the three characters indicate permission to read, to write, and to execute the file as a command, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for any included files or directories. In summary, these permissions are indicated as follows:

r if the file is readable

w if the file is writable

x if the file is executable

\- if the indicated permission is not granted

Changing Read, Write, and Execute Permissions

To change a file or directory's read, write, and execute permissions, you need only use one command: **chmod**. Several examples are included here to illustrate the syntax for this command. Remember that there are three distinct classes of users: user, group, and other, and that for each class all three permissions are either set or not set.

For example, presume that a file named filel exists with the following permissions:

```
-rw-r-----
```

To give filel read permission for all classes of users,

type:

```
chmod a+r filel
```

Here, "a" stands for "all," which is a synonym for "ugo" where "u" stands for user, "g" for group, and "o" for other users. If no user class is specified, then "a" is normally the default, although this default may vary. The resulting permissions are:

```
-rw-r--r--
```

To give write and execute permissions to members of a group only, type:

```
chmod g+wx filel
```

For a filel with the attributes

```
-rw-------
```

the above command would alter attributes so that they looked like this:

```
-rw--wx---
```

To remove write and execute permission by the user (owner) and group associated with filel, type:

```
chmod ug-wx filel
```

Note how letters are grouped to set specific user/permission combinations.

Changing Directory Search Permissions

Obviously, directories cannot be executable files, yet they, also, have an execute attribute that can be set. This attribute signifies search permission. if this permission is denied to a particular user, then that user cannot even list the names of the files in the directory.

For example, assume that the directory dirl has the following attributes:

```
drwxr-xr-x
```

To remove search permission for other users to examine dirl, type:

```
chmod o-xr dirl
```

The new attributes for dirl are then:

```
drwxr-x---
```

Note that the read permission for the directory was also removed with the above chmod command. In the great majority of cases you will want to treat these two permissions as a pair when changing directory permissions.

4.9 Information Processing

Calculating

The **bc** command invokes an interactive desk calculator that can be used as if it were a hand-held calculator. A typical session with **bc** is shown below. Comments show what action is performed after each input line:

```
/* This is a comment    */
123.456789 + 987.654321 /* Add and output      */
1111.111110
9.0000000 - 9.0000001   /* Subtract and output */
-.0000001
64/8                    /* Divide and output   */
8
1.12345678934 * 2.3     /* Note precision      */
2.58395061548
19%4                    /* Find remainder      */
3
3^4                     /* Exponentiation      */
81
2/1*2                   /* Note precedence     */
4
2/(1*2)                 /* Note precedence again */
1
x = 46.5                /* Assign value to x   */
y = 52.5                /* Assign value to y   */
x + y + 1.0000          /* Add and output      */
100.0000
obase=16                /* Set hex output base */
15                      /* Convert to hex      */
F
16                      /* Convert to hex      */
10
64                      /* Convert to hex      */
40
255                     /* Convert to hex      */
FF
256                     /* Convert to hex      */
100
512                     /* Convert to hex      */
200
quit                    /* Must type whole word */
```

Also available are scaling, function definition, and programming statements much like those in the C programming language. Other features include assignment to named registers and subroutine calling. For more information, see Chapter 9, BC: A Calculator.

Echoing Arguments

The echo command is used to echo arguments to the standard output. For example

```
echo hello
```

outputs

```
hello
```

To output several lines of text, surround the echoed argument in double quotes. For example, type:

```
echo "Now is the time\
For all good men\
To come to their party."
```

This will output:

```
Now is the time
For all good men
To come to their party.
```

One use for echo is to remind yourself of important events. For example, you might type

```
echo "Meeting at 9:00am." | mail self
```

where "self" is your log in name. The above example pipes your message into mail; mail then sends you the reminder. Echo is also particularly useful if you should ever program in the shell command language. For more information about the shell, see Chapter 7, "The Shell".

Sorting A File

One of the most generally useful file processing commands is sort. By default, sort sorts the lines of a file according to the ASCII collating sequence. For example, to sort a file named phonelist, type:

```
sort phonelist
```

In the above case, the sorted contents of the file are displayed on the screen. To create a sorted version of phonelist named phonesort, type:

```
sort phonelist >phonesort
```

Note that sort is useful in sorting the output from other commands. For example, to sort the output from execution of a who command, type:

```
who | sort >whosort
```

A wide variety of options are available for sort. For more information, see sort(1) in the XENIX Reference

Manual.

Searching for a Pattern in a File

The **grep** command selects and extracts lines from a file, printing only those lines that match a given pattern. The name **grep** is an acronym for **g**lobal regular expression **p**rint, Grep is one of the most useful commands in all of XENIX, so learn it well.

For example, to print out all lines in a file containing the word "tty38", type:

```
grep 'tty38' file
```

In general, you should always enclose the pattern you are searching for in single quotes ('), so that special metacharacters are not expanded unexpectedly by the shell.

Grep is also useful in conjunction with another command to "filter" the output from one command, removing all unwanted lines.

As another example, assume that you have a file named phonelist that contains on each line a name followed by a phone number. Assume also that there are several thousand lines in this list. What is the quickest way to find the phone number of someone named joe, whose phone number prefix is 822? The answer, of course, is to use **grep**:

```
grep 'joe' phonelist | grep '822-' >joes.number
```

What happens above is that **grep** finds all occurrences of lines containing the word "joe" in the file phonelist. The output from this command is then filtered through another **grep** command, which searches for an "822-" prefix, thus removing any unwanted joes. Finally, assuming that a unique phone number for joe exists with the "822-" prefix, that name and number are placed in the file joes.number.

For more information about **grep**, and the types of patterns it can be used to search for (called regular expressions) see grep(1) and its relatives, fgrep(1), and egrep(1) in the XENIX Reference Manual.

Counting Words, Lines, and Characters

Wc is a simple, useful, easy-to-use utility. The letters "wc" stand for word count. Thus, **wc** counts the number of words in a file. Words are presumed to be separated by punctuation, spaces, tabs, or new lines. **Wc** also counts characters and lines; all three counts are reported by default. For example, to count the number of lines, words, and characters in the file textfile, type:

```
wc textfile
```

Typical output might be:

```
  4432  18188   97808 textfile
```

To specify a count of characters, words, or lines only, you must use an appropriate mnemonic switch. To illustrate, examine the following three commands and the output produced by each:

```
wc -c textfile
  97808 textfile

wc -w textfile
  18188 textfile

wc -l textfile
   4432 textfile
```

The first example prints out the number of characters in textfile, the second prints out the number of words, and the third prints out the number of lines.

4.10 Line Printers

Sending a File to the Line Printer

One of the most common operations that you will want to perform is printing files on the line printer. The most straightforward method for doing this is to type

```
lpr file1
```

for one file, or

```
lpr file1 file2 file3
```

for multiple files. Other common uses of **lpr** involve pipes. For instance, to paginate and print a file of raw text, type:

```
pr textfile | lpr
```

The **pr** and **lpr** commands are very often used together. As another example, to sort, paginate, and print a file, type:

```
sort datafile | pr | lpr
```

Getting Line Printer Queue Information

More than one file may be waiting to be printed at the line printer: XENIX does not require that the file be printed before the **lpr** command finishes. Instead, **lpr** makes sure only that the file is placed in a special directory where it will wait its turn to be printed.

The files in this queue are specified by /usr/spool/lpd/[cl]f*. Thus, to examine the line printer queue, type:

```
ls -l /usr/spool/lpd/[cl]f*
```

4.11 Communications

Sending Mail

Mail is a system-wide facility that permits you and other system users to send and receive mail. To send mail to another user on the system, type

```
mail joe
```

where joe is the name of any user of the system. Following entry of the command, you enter the actual text of the message you want to send. Entry of text is terminated by typing a <CONTROL-D>. A complete session at the terminal might go like this:

```
mail joe
Subject: Meeting
There will be a meeting at 2:00 today
to review recent problems with the
new system.
<CONTROL-D>
```

For practice, send mail to yourself. (This isn't as strange as it might sound -- mail to oneself is a handy reminder mechanism.)

You can also send a previously prepared letter, and you can send mail to a number of people all at once. For more details see mail(1) in the XENIX Reference Manual.

Receiving Mail

When you log in, you may sometimes get the message:

```
You have mail.
```

To read your mail, type:

```
mail
```

A heading for each message is then displayed on your terminal screen. After typing <RETURN>, the contents of the first message are displayed. Subsequent messages are displayed, one message at a time, most recent message first, each time you type <RETURN>.

After each message is displayed, **mail** waits for you to say what to do with it. The two basic responses are d, which deletes the message, and <RETURN>, which does not delete the message (so it will still be there the next time you read your mailbox). To exit mail, type q, for "quit." Other responses are described in the XENIX Reference Manual under mail(1).

Using The Automatic Reminder Service

An automatic reminder service is normally available for all XENIX users. Once each day, XENIX uses the **calendar** command to examine each user's home directory for a file named calendar, the contents of which might look something like this:

```
1/23 David's wedding
2/9  Mira's birthday
4/27 Meeting at 2:00
9/1  Karen's birthday
10/3 License renewal
```

Calendar examines each line of the calendar file, culling from the file those lines containing today's and tomorrow's dates. These lines are then mailed to you to remind you of the specified events.

Writing To A Terminal

To write directly to another user's terminal, use the write command. For example, to write to joe's terminal, type:

```
write joe
```

After you have executed the command, each subsequent line that you type is displayed both on your own terminal screen and on joe's. To terminate writing of text to joe, enter a <CONTROL-D>. The procedure for a two-way write is for each party to end each message with a distinctive signal, normally (o) for "over." The signal (oo) for "over and out" is used when a conversation is about to be terminated.

CHAPTER 5

ED

CONTENTS

5.1 Introduction

Ed is a text editor used to create and modify text. The text is normally a document, a program, or data for a program. The recommended way to learn ed is to read this chapter, simultaneously using ed to follow the examples. No matter what, you should do the exercises; they cover material not completely discussed in the actual text.

This chapter is organized so that the most basic information is presented early on. More difficult and advanced topics are discussed in later sections. A complete summary of all ed commands appears as the last section in the chapter.

5.2 Entering and Exiting The Editor

The simplest way to invoke ed, is to type:

```
ed
```

The most common way, however, is to type:

```
ed filename
```

where filename is the name of a new or existing file.

To exit the editor, all you need to do is type:

```
q
```

If you have not written out any changes you have made before you quit, ed will warn you that you will lose these changes by printing a question mark (?). If you still want to quit, type another **q**.

5.3 Commands

The following subsections describe the individual ed commands and how they can be used. Note that most commands are mnemonically named to describe their function.

5.3.1 Append - a

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper. Clearly, it will have to start somewhere and undergo modifications later. This section shows you how to put text in a file, just to get started.

Later we'll talk about how to change it.

When ed is first started, it is like working with a blank piece of paper -- there is no text or information present. This must be supplied by the person using ed: it is usually done by typing in the text, or by reading it into ed from a file. We will start by typing in some text, and discuss how to read files later.

First a bit of terminology. In ed jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space if you like, or simply as the information that you are going to be editing. In effect, the buffer is the piece of paper on which you will write things, make changes, and finally file away for another day.

You tell ed what to do to your text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. Ed makes no response to most commands;there is no prompting or typing of messages.

The first command is "append", written as the letter "a" on a line by itself. It means "append (or add) text lines to the buffer, as they are typed in." Appending is like writing fresh material on a piece of paper.

To enter lines of text into the buffer, just type an "a", followed by a <RETURN>, followed by the lines of text you want, like this:

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

To stop appending, type a line that contains only a period on a line by itself. A period (.) is used to tell ed that you have finished appending. (You can also use <CONTROL-D>, but we will use the period throughout this discussion.) If ed seems to be ignoring you, type an extra line with just a period (.) on it. You may then find you've added some garbage lines to your text, which you will have to take out later.

After the append has completed, the buffer will contain the following three lines:

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

The a and . aren't there, because they are not text.

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

If at any time you make an error in the commands you type to ed, it will tell you by typing:

```
?
```

With practice, you should usually be able to figure out how you have erred.

5.3.2 Write - w

It's likely that you will want to save your text for later use. To write out the contents of the buffer into a file, use the "write command, **w**, followed by the name of the file that you want to write to. This copies the buffer's contents to the specified file, destroying any previous information in the file. For example, to save the text in a file named text, type:

```
w text
```

Leave a space between **w** and the filename. Ed responds by printing the number of characters it has written out. For instance, ed might respond with

```
68
```

(Remember that blanks and the new line character at the end of each line are included in the character count.) Writing a file just makes a copy of the text -- the buffer's contents are not disturbed, so you can go on adding lines to it.

This is an important point. Ed at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. Writing out the text to a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer. However, any text that was written out to a file is relatively safe.

5.3.3 Quit - q

To terminate a session with ed, save the text you're working on by writing it to a file using the w command, and then type:

```
q
```

The system responds with the XENIX prompt character. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting. Actually, ed will print "?", if you try to quit without writing. At that point, write out the text if you want; if not, typing another q will get you out.

Exercise

Enter ed and create some text by typing:

```
a
... text ...
.
```

Write it out by typing:

```
w filename
```

Then leave ed by typing:

```
q
```

Next use the cat command to display the file on your terminal screen to see that everything has worked.

5.3.4 Edit - e

A common way to get text into your editing buffer is to read it in from a file. This is what you do to edit text that you have saved with the w command in a previous session. The edit command e fetches the entire contents of a file to the buffer. So if you had saved the three lines "Now is the time", etc., with a w command in an earlier session, the ed command

```
e text
```

would fetch the entire contents of the file text into the buffer and might respond with

68

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

If you use the e command to read a file into the buffer, then you need not use a filename after a subsequent w command. Ed remembers the last filename used in an e command, and w will write to this file. Thus, a good way to operate is this:

```
ed
e file
[editing session]
w
q
```

This way, you can type w from time to time and be secure in the knowledge that if you got the filename right in the beginning, you are writing out to the proper file each time.

5.3.5 File - f

You can find out the remembered filename at any time with the "file" command, f. Just type f without a filename. You can also change the name of the remembered filename with f; a useful sequence is

```
ed precious
f  junk
   [rest of editing session]
```

which gets a copy of a precious file, then uses f to guarantee that a careless w command won't overwrite the original.

5.3.6 Read - r

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done with the "read" command, r. For example, the command

```
r text
```

reads the file text into your editing buffer and adds it to the end of whatever is already in the buffer. As another example, pretend that you have performed a read after an edit, like so:

```
e text
r text
```

The buffer will then contain two copies of text (six lines):

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

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

Exercise

Experiment with the e command -- try reading and printing various files. You may get the error message

```
?name
```

where name is the name of a nonexistent file. This means that the file doesn't exist, typically because you spelled the filename wrong, or perhaps because you are not allowed to read from or write to the file. Try alternately reading and appending to see that they work similarly. Verify that

```
ed file.text
```

is equivalent to

```
ed
e file.text
```

5.3.7 Print - p

To print the contents of the buffer (or parts of it) on the terminal screen, use the "print" command, **p**. To do this, specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter "p". Thus, to print the first two lines of the buffer, (that is, lines 1 through 2) type:

```
1,2p
```

Ed responds with:

```
Now is the time
for all good men
```

Suppose you want to print all the lines in the buffer. You could use 1,3p as above if you knew there were exactly 3 lines in the buffer. But you will rarely know how many lines there are, so what do you use for the final line number? The answer is that ed provides a shorthand symbol for "the line number of the last line in the buffer" -- the dollar sign ($). Use it this way:

```
1,$p
```

This will print all the lines in the buffer (from line 1 to the last line). If you want to stop the printing before it is finished, press the <INTERRUPT> key. Ed then types

```
?
```

and waits for entry of the next ed command.

To print the last line of the buffer, use:

```
$p
```

You can print any single line by typing the line number, followed by a p. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, ed lets you abbreviate even further: you can print any single line by typing just the line number; there's no need to type the letter **p**. So if you type

```
$
```

ed prints the last line of the buffer.

You can also use $ in combinations like

```
$-1,$p
```

which prints the last two lines of the buffer. This helps when you want to see how far you are in your typing.

The next step is to use address arithmetic to combine the line numbers like dot (.), dollar ($), with plus (+) and minus (-). Thus

```
$-1
```

is a command to print the next to last line of the current file (that is, one line before the line dollar($)). For example, to recall how far you got in a previous editing session

```
$-5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, you'll get an error message.

As another example

```
.-3,.+3p
```

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the plus (+) can be omitted: For example

```
.-3,.3p
```

is identical in meaning.

Another area in which you can save typing effort in specifying lines is to use plus and minus as line numbers by themselves. For example

```
-
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines. For example

```
---
```

moves up three lines, as does "-3". Thus

```
-3,+3p
```

is also identical to the examples above.

Exercise

As before, create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print lines in reverse order by saying 3,1p, do not work.

5.3.8 Current Line- dot (.)

Suppose your editing buffer still contains the following six lines:

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

If you type

```
1,3p
```

ed will display

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

Now try typing:

```
p
```

This prints

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. You can repeat this **p** command without line numbers, and ed will continue to print line 3.

This happens because ed maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. The line most recently acted on is referred to with a period (.) and is called "dot." Dot is a line number in the same way that dollar ($) is; it means "the current line," or loosely, "the line you most recently did something

to." You can use it in several ways. One possibility is to type:

```
.,$p
```

This will print all the lines from (and including) the current line clear to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed. In the example above, **p** sets both dot and $ to 6.

Dot is often used in combinations like this one:

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

This means "print the next line" and is one way of stepping slowly through the editing buffer. You can also type

```
.-1     (or .-1p)
```

which means "print the line before the current line." This enables you to go backwards through the file if you wish. Another useful command is something like

```
.-3,.-1p
```

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing:

```
.=
```

Ed responds by printing the value of dot.

Let's summarize some things about the **p** command and dot. Essentially, p can be preceded by 0, 1, or 2 line numbers. If there is no line number given, ed prints the "current line," the line that dot refers to. If there is one line number given (with or without the letter **p**), ed prints that line (and dot is set there); and if there are two line numbers, ed prints all the lines in that range (and sets dot to the last line printed). If two line numbers are specified the first cannot be bigger than the second.

Typing a single <RETURN> causes printing of the next line. It is equivalent to:

```
.+1p
```

Try it. Next, try typing a minus sign (-) alone; you will find that it is equivalent to typing:

```
.-1p
```

5.3.9 Delete - d

Suppose you want to get rid of the three extra lines in the buffer. This is done with the "delete" command, **d**. Its action is similar to that of p, except that d deletes lines instead of printing them. The lines to be deleted are specified for **d** exactly as they are for **p**:

starting-line,ending-lined

Thus, the command

```
4,$d
```

deletes lines 4 through the end. There are now three lines left in our example, as you can check by typing:

```
1,$p
```

Notice that $ now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Exercise

Experiment with a, e, r, **w**, **p**, and d until you are sure that you know what they do, and until you understand how dot (.), dollar ($), and line numbers are used.

If you are adventurous, try using line numbers with a, r, and **w**, as well. You will find that a appends lines after the line number that you specify (rather than after dot); that r reads in a file after the line number you specify (not necessarily at the end of the buffer); and that **w** writes out exactly the lines you specify, but not necessarily the whole buffer. These variations are sometimes handy. For instance, you can insert a file at the beginning of a buffer by typing

0r filename

and you can enter lines at the beginning of the buffer by

typing:

```
0a
[input text here]
.
```

Notice that typing

```
.w
```

is very different from

```
.
w
```

since the former writes out only a single line, and the latter writes out the whole file.

5.3.10 Substitute - s

We are now ready to try one of the most important of all commands -- the "substitute" command s. This is the command that is used to change individual words or letters within a line or group of lines. It is what you use for correcting spelling mistakes and typing errors.

Suppose that, due to a typing error, line 1 says:

```
Now is th time
```

The letter "e" has been left off of the word "the". You can use s to fix this up as follows:

```
1s/th/the/
```

This says to substitute for the characters "th", the characters "the", in line 1. To verify that the substitution has worked, type

```
p
```

to get

```
Now is the time
```

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

The syntax for the substitute command follows:

starting-line,ending-lines/pattern/replacement/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in all the lines between starting-line and ending-line. Only the first occurrence on each line is changed, however. How to change every occurrence is discussed later in the section. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution takes place, dot is not changed. This causes printing of a question mark (?) as a warning.

Thus, you can type

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text.

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

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, you can try again. (Notice that there is a p on the same line as the s command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.)

It's also legal to type

```
s/string//
```

which means "change the first string of characters to nothing," or, in other words, remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

```
Nowxx is the time
```

you could type

```
s/xx//p
```

to get

```
Now is the time
```

Notice that two adjacent slashes mean "no characters," not a space. There is a difference!

Exercise

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

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

This results in:

```
on the other side of the coin
```

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

```
s/ ... / ... /gp
```

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

5.3.11 Search - /string/ and ?string?

Now that you've mastered the substitute command, you can move on to mastering another important concept: context searching.

Suppose you have the original three line text in the buffer:

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

Suppose you want to find the line that contains the word "their", so that you can change it to the word "the.". With only three lines in the buffer, it's pretty easy to keep track of which line the word "their" is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would

be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some textual pattern contained on the line.

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

```
/string of characters we want to find/
```

For example, the ed command

```
/their/
```

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

```
to come to the aid of their party.
```

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

```
?
```

Otherwise, ed prints the line it found.

You can also search backwards in a file for search strings by using question marks instead of slashes. For example

```
?thing?
```

searches backwards in the file for the word "thing" This is especially handy when you realize that the thing you want to operate on is backwards from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command. (If you get unexpected results using any of the characters

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

read "Section 5.4, Context and Regular Expressions.")

You can do both the search for the desired line and a substitution all at once, like this:

```
/their/s/their/the/p
```

This yields:

```
to come to the aid of the party.
```

There are three separate actions in the last command. The first is a context search for the desired line, the second is the substitution, and the third is the printing of the line.

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

Suppose the buffer contains the three familiar lines

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

Then the ed line numbers

```
/Now/+1
/good/
/party/-1
```

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

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. For instance, you could print all three lines by typing either

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or by any number of similar combinations. The first combination is better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use

```
1,$p
```

but not if there were several hundred.)

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

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the "horrible thing" that you wanted, so it is necessary to repeat the search again. You don't have to retype the search, for the construction

```
//
```

is a shorthand for "the previous thing that was searched for," whatever it was. This can be repeated as many times as necessary. You can also go backwards, since

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use // as the left side of a substitute command, to mean "the most recent pattern." For example, examine:

```
/horrible thing/
[ed prints line with "horrible thing"]
s//good/p
```

To go backwards and change "horrible thing" to "good", type:

```
??s//good/
```

Exercise

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

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

Try context searching using ?text? instead of /text/. This scans lines in the buffer in reverse order rather than normal order. This is sometimes useful if you go too far while looking for a string of characters. It's an easy way to back up in the file you're editing.

(If you get unexpected results with any of the characters

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

read "Section 5.4, Context and Regular Expressions.")

5.3.12 Change and Insert - c and i

This section discusses the "change" command, **c**, which is used to change or replace a group of one or more lines, and the "insert" command, i, which is used for inserting a group of one or more lines.

The **c** command is used to replace a number of lines with different lines that you type at the terminal. For example, to change lines .+1 through $ to something else, type

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

The lines you type between the **c** command and the dot (.) will take the place of the original lines between start line and end line. This is useful in replacing a line or several lines that have errors in them.

If only one line is specified in the c command, then only that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of a period to end the input. This works just like the period in the append command and must appear by itself on a new line. If no line number is given, the current line specified by dot is replaced. The value of dot is set to the last line you typed in. Note that the terminating period and the line

referenced by dot are completely different: the first is used simply to terminate a command, the second points at a specific line of text.

The **i** command is similar to the append command. For example

```
/string/i
[type the lines to be inserted here ...]
.
```

inserts the given text before the next line that contains "string". The text between **i** and the terminating period is inserted before the specified line. If no line number is specified, dot is used. Dot is set to the last line inserted.

Exercise

The c command is like a combination of delete followed by insert. Experiment to verify that

```
start,endd
i
[text]
.
```

is almost the same as

```
start,endc
[text]
.
```

These are not precisely the same if line dollar ($) gets deleted.

Experiment with **a** and **i** to see that they are similar, but not the same. Observe that

```
line-numbera
[text]
.
```

appends after the given line, while

```
line-numberi
... text ...
.
```

inserts before it. Observe that if no line number is given, i inserts before line dot, while a appends after line dot.

5.3.13 Move - m

The "move" command, m, is used for cutting and pasting. It lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by typing

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

where temp is the name of a temporary file. However, you can do it more easily with the m command:

```
1,3m$
```

The general case is

start-line,end-linemafter-this-line

Notice that there is a third line to be specified: the place where the moved text gets put. Of course, the lines to be moved can be specified by context searches. If you had

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

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/-1
```

Notice the -1. The moved text goes after the line mentioned. Dot gets set to the last line moved.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to one line after line dot. If you are positioned on the second line,

```
m--
```

does the interchange.

The m command is more succinct and direct than writing, deleting and rereading. This is a matter of personal taste; do what you have most confidence in. The main difficulty with the m command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may not move the lines you wanted. The result of a bad m command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted. It is also a good idea to issue a w command before doing anything complicated; then if you make a mistake, it's easy to back up to where you were.

5.3.14 Global - g and v

The "global" commands g and v are used to execute one or more editing commands on all lines that either contain (g) or don't contain (v) a specified pattern.

As the simplest example, the command

```
g/XENIX/p
```

prints all lines that contain the word "XENIX". The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with ".").

The v command is identical to g, except that it operates on those line that do not contain an occurrence of the pattern. Mnemonically, the "v" can be thought of as part of the word inverse.

For example

```
v/^\./p
```

prints all the lines that don't begin with a period (i.e., the actual text lines).

Any command can follow g or v. For example, the following command deletes all lines that begin with ".":

```
g/^\./d
```

This command deletes all empty lines:

```
g/^$/d
```

Probably the most useful command that can follow a global command is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word "Xenix" to "XENIX" everywhere, and verify that it really worked, with

```
g/Xenix/s//XENIX/gp
```

Notice that we used /.../ in the substitute command to mean "the previous pattern," in this case, "Xenix". The p command executes on each line that matches the pattern, not just on those in which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line is examined in turn, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** command to use addresses, set dot, and so on, quite freely. For example:

```
g/^\.P/+
```

prints the line that follows each ".P" command (the signal for a new paragraph in some formatting packages). Remember that plus (+) means "one line past dot." And

```
g/topic/?^\.H?1
```

searches for each line that contains "topic", scans backwards until it finds a line that begins ".H" (a heading) and prints the line that follows that, thus showing the headings under which "topic" is mentioned. Finally

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with ".EQ" and ".EN" formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

It is possible to give more than one command under the control of a global command. As an example, suppose the

task is to change "x" to "y" and "a" to "b" on all lines that contain "thing". Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The backslash (\) signals the g command that the set of commands continues on the next line; it terminates on the first line that does not end with a backslash.

Note that you cannot use a substitute command to insert a new line within a g command.

You should watch out for this problem. The command

```
g/x/s//y/\
s/a/b/
```

does not work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be "x" (as expected), and sometimes it will be "a" (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and i commands as part of a global command. As with other multi-line constructions, all that is needed is to add a backslash at the end of each line except the last. Thus, to add a ".nf" and ".sp" command before each ".EQ" line, type:

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a period (.) to terminate the i command, unless there are further commands to be executed under the global command. On the other hand, it does no harm to put it in either.

5.3.15 List - l

Ed provides two commands for printing the contents of the lines you're editing. You should be familiar with **p**, in combinations like

```
1,$p
```

to print all the lines you are editing, or

```
s/abc/def/p
```

to change "abc" to "def" on the current line. Less familiar is the list command l, which gives slightly more information than p. In particular, l makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, l prints each tab as ">" and each backspace as "<". This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The l command also "folds" long lines for printing. Any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash (\), so you can tell it was folded. This is useful for printing lines longer than the width of your terminal screen.

Occasionally, the l command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each backslash-number combination is a single character. When you see such characters, be wary: they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

5.3.16 Undo - u

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. The "undo" command, u, lets you "undo" the last substitution: the last line that was substituted can be restored to its previous state by typing:

```
u
```

This command does <u>not</u> work with the g and v commands.

5.3.17 **Mark - k**

The mark command, **k**, provides a facility for marking a line with a particular name, so that you can later reference it by name, regardless of its actual line number. This can be handy for moving lines and keeping track of them as they move. For example

```
kx
```

marks the current line with the name "x". If a line number precedes the **k**, that line is marked. (The mark name must be a single lowercase letter.) Now you can refer to the marked line with the notation:

```
'x
```

Note the use of the single quote (') here. Marks are most useful for moving things around. Find the first line of the block to be moved and then mark it with:

```
ka
```

Then find the last line and mark it with

```
kb
```

Now position yourself at the place where the text is to go and type:

```
'a,'bm.
```

Bear in mind that a line can have only one mark name associated with it at any given time.

5.3.18 **Transfer - t**

We mentioned earlier the idea of saving a line that is hard to type or used often, to cut down on typing time. Of course, this could be more than one line; then the saving is presumably even greater.

Ed provides another command, called t (for transfer) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The t command is identical to the m command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing.

A more common use for t is to create a series of lines that differ only slightly. For example, you can type

```
a
abcdefghijkl  x  zzzzzzz [long line]
.
t.       [make a copy]
s/x/y/   [change it a bit]
t.       [make third copy]
s/y/z/   [change it a bit]
```

and so on.

5.3.19 Shell Escape - !

Sometimes it is convenient to temporarily escape from the editor to execute a XENIX command without leaving the editor. The shell "escape" command, !, provides a way to do this. You can really execute any XENIX command, including another ed. (This is quite common, in fact.)

If you type

```
!command
```

your current editing state is suspended, and the XENIX command you asked for is executed. When the command finishes, ed will signal you by printing another exclamation (!); at that point you can resume editing.

5.4 Context and Regular Expressions

You may have noticed that things don't work right when you use characters such as the period (.), the asterisk (*), the dollar sign ($), and others in context searches and the substitute command. The reason is rather complex, although the solution to the problem is simple. Ed treats these characters as special. For instance, in a context search or the first string of the substitute command, the period (.) means "any character," not a period, so

```
/x.y/
```

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

a line with an "x", a period, and a "y". A complete list of the special characters that can cause trouble follows:

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

The next few subsections will discuss how to use these characters to describe patterns of text in search and substitute commands. These patterns are called "regular expressions", and occur in several other important XENIX commands and utilities, including grep(1), fgrep(1), egrep(1), sed(1), sh(1), ex(1), and vi(1).

As the simplest place to begin, recall the meaning of a trailing g after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the first "this" on the line with "that". If there is more than one "this" on the line, the second form with the trailing g changes all of them.

Either form of the s command can be followed by p or l to print or list the contents of the line. For example, all of the following are legal and mean slightly different things:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

Make sure you know what the differences are.

Of course, any s command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the first occurrence of "mispell" to "misspell" in each line of the file. But

```
1,$s/mispell/misspell/g
```

changes every occurrence in each line (and this is more likely to be what you wanted).

You should also notice that if you add a p or l to the end of any of these substitute commands, only the last line changed is printed, not all the lines. We will talk later about how to print all the lines that were modified.

5.4.1 Period - (.)

The first metacharacter that we will discuss is the period (.). On the left side of a substitute command, or in a search, a period (.) stands for any single character. Thus the search

```
/x.y/
```

finds any line where "x" and "y" occur separated by a single character, as in

```
x+y
x-y
x y
xzy
```

and so on.

Since a period matches a single character, it gives you a way to deal with funny characters printed by l. Suppose you have a line that appears as

```
th\07is
```

when printed with the l command, and that you want to get rid of the \07, which represents an ASCII bell character.

The most obvious solution is to try

```
s/\07//
```

but this will fail. The common solution, which most people would now take, is to retype the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big. But for a very long line, retyping is not the best solution. This is where the metacharacter "." comes in handy. Since \07 really represents a single character, if we type

```
s/th.is/this/
```

the job is done. The "." matches the mysterious character between the "h" and the "i", whatever it is.

Bear in mind that since "." matches any single character, the command

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

converts the first character on a line into a comma (,), which very often is not what you intended.

As is true of many characters in ed, the period (.) has several meanings, depending on its context. This line shows all three:

```
.s/././
```

The first period is the line number of the line we are editing, which is called "dot." The second period is a metacharacter that matches any single character on that line. The third period is the only one that really is an honest, literal period. (Remember also that a period is also used to terminate input from the a and i commands.) On the right side of a substitution, the period (.) is not special. If you apply this command to the line

```
Now is the time.
```

the result is

```
.ow is the time.
```

which is probably not what you intended.

5.4.2 **Backslash - **

Since a period means "any character," the question naturally arises: what do you do when you really want a period? For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash (\) does the job. A backslash turns off any special meaning that the next character might have; in particular, "\." converts the "." from a "match anything" into a literal period, so you can use it to replace the period in "Now is the time." like this:

```
s/\./?/
```

The pair of characters "\." is considered by ed to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.DE
```

at the start of a line. The search

```
/.DE/
```

isn't adequate, for it will find lines like

```
JADE
FADE
MADE
```

because the "." matches the letter "A" on each of the lines in question. But if you type

```
/\.DE/
```

only lines that contain ".DE" are found.

The backslash can be used to turn off special meanings for characters other than the period. For example, consider finding a line that contains a backslash. The search

```
/\/
```

won't work, because the backslash (\) isn't a literal backslash, but instead means that the second slash (/) no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus, this works:

```
/\\/
```

Similarly, you can search for a forward slash (/) with

```
/\//
```

The backslash turns off the special meaning of the immediately following "/" so that it doesn't terminate the slash-slash construction / / prematurely.

As an exercise, find two substitute commands each of which convert the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; you should verify that each works:

```
s/\\\.//
s/x../x/
s/..y/y/
```

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an s command; there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter. To delete all the slashes, type

```
s:/::g
```

Second, if "#" and "@" are your character erase and line kill characters, you have to type \# and \@; this is true whether you're talking to ed or to any other program.

When you are adding text with a or i or c, backslash is not special, and you should only put in one backslash for each one you really want.

5.4.3 Dollar Sign - $

The next metacharacter, the "$", stands for "the end of the line." As its most obvious use, suppose you have the line

```
Now is the
```

and you wish to add the word "time" to the end. Use the dollar sign ($) like this

```
s/$/ time/
```

to get:

Now is the time

Notice that a space is needed before "time" in the substitute command, or you will get:

Now is thetime

As another example, replace the second comma in the following line with a period without altering the first:

Now is the time, for all good men,

The command needed is:

s/,$/./

The dollar sign ($) here provides context to make specific which comma we mean. Without it, of course, the s command would operate on the first comma to produce:

Now is the time. for all good men,

To convert:

Now is the time.

into

Now is the time?

as we did earlier, we can use:

s/.$/?/

Like the period (.), the dollar sign ($) has multiple meanings depending on context. In the line

$s/$/$/

the first "$" refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

5.4.4 Circumflex - ^

The circumflex or caret (^) stands for the beginning of the line. For example, suppose you are looking for a line that begins with "the". If you simply type

```
/the/
```

you will in all likelihood find several lines that contain "the" in the middle before arriving at the one you want. But with

```
/^the/
```

you narrow the context, and thus arrive at the desired line more easily.

The other use of the circumflex (^) enables you to insert something at the beginning of a line. For example

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains only the characters

```
.P
```

you can use the command

```
/^\.P$/
```

5.4.5 Star - *

Suppose you have a line that looks like this:

```
text x     y text
```

where "text" stands for lots of text, and there are some indeterminate number of spaces between the "x" and the "y". Suppose the job is to replace all the spaces between "x" and "y" by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter "star" (*) comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, type:

```
s/x *y/x y/
```

The construction " *" means "as many spaces as possible." Thus "x *y" means an "x", as many spaces as possible, then a "y".

The star can be used with any character, not just a space. If the original example was

```
text x--------y text
```

then all minus signs (-) can be replaced by a single space with the command:

```
s/x-*y/x y/
```

Finally, suppose that the line was:

```
text x..................y text
```

Can you see what trap lies in wait for the unwary? If you blindly type

```
s/x.*y/x y/
```

what happens? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good thinking. Remember that "." matches any single character? Then ".*" matches as many single characters as possible, and unless you are careful, it can eat up a lot more of the line than you expected. For example, if the line was like this

```
x text x.........y text y
```

then typing

```
s/x.*y/x y/
```

takes everything from the first "x" to the last "y", which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of the period (.) with the backslash (\):

```
s/x\.*y/x y/
```

Now everything works, for "\.*" means "as many periods as possible".

There are times when the pattern ".*" is exactly what you want. For example, to change

```
Now is the time for all good men ....
```

into

```
Now is the time.
```

use ".*" to eat up everything after the "for":

```
s/ for.*/./
```

There are a couple of additional pitfalls associated with star (*) that you should be aware of. Most notable is the fact that "as many as possible" means zero or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

```
xy text x  y text
```

and we said

```
s/x *y/x y/
```

the first "xy" matches this pattern, for it consists of an "x", zero spaces, and a "y". The result is that the substitute acts on the first "xy", and does not touch the later one that actually contains some intervening spaces.

The way around this is to specify a pattern like

```
/x  *y/
```

which says an "x", a space, then as many more spaces as possible, then a "y", in other words, one or more spaces.

The other startling behavior of star (*) again relates to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legitimate number of matches, and there are no x's at the beginning of the line (so that gets converted into a "y"), nor between the "a" and the "b" (so that gets converted into a "y"), and so on. Make sure you really want zero matches; if not, in this case write

```
s/xx*/y/g
```

since "xx*" is one or more x's.

5.4.6 Brackets - [and]

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. That is the purpose of brackets ([and]).

The construction

```
[0123456789]
```

matches any single digit -- the whole thing is called a "character class." With a character class, the job is easy. The pattern "[0123456789]*" matches zero or more digits (an entire number), so

```
1,$s/^[0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and there are only three special characters (^,], and -) inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can type:

```
/[.\$^[]/
```

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0-9]; similarly, [a-z] stands for the lowercase letters, and [A-Z] for uppercase.

Within [...], the "[" is not special. To get a "]" (or a "-") into a character class, make it the first character.

As a final note on character classes, you can specify a class that means "none of the following characters." This is done by beginning the class with a circumflex (^). For example

```
[^0-9]
```

stands for "any character except a digit". Thus, you might find the first line that doesn't begin with a tab or space with a search like:

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^^]/
```

finds a line that doesn't begin with a circumflex.

5.4.7 Ampersand - &

To save typing, the ampersand symbol is used in substitutions. Suppose you have the line

```
Now is the time
```

and you want to make it:

```
Now is the best time
```

Of course you can always type:

```
s/the/the best/
```

It's unnecessary to repeat the word "the". The ampersand (&) is used to eliminate this repetition. On the right side of a substitute, the ampersand means "whatever was just matched," so you can type

```
s/the/& best/
```

and the ampersand will stand for "the". Of course this isn't much of a saving if the thing matched is just "the", but if it is something very long, or if it is something like ".*" which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length, type:

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

The ampersand can occur more than once on the right side. For example

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

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

converts the original line into

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

To get a literal ampersand, the backslash is used to turn off the special meaning. For example

```
s/ampersand/\&/
```

converts the word into the symbol. The ampersand is not special on the left side of a substitute command, only on the right side.

5.4.8 Substituting New Lines

Ed provides a facility for splitting a single line into two or more shorter lines by "substituting in a new line." As the simplest example, suppose a line has become unmanageably long because of editing. If it looks like

```
text xy text
```

you can break it between the "x" and the "y" like this:

```
s/xy/x\
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that the backslash (\) turns off special meanings, it seems relatively intuitive that a backslash at the end of a line would make the new line there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word "very" in a long line by splitting "very" onto a separate line, and preceding it by the formatting command ".I":

```
text a very big text
```

The command

```
s/ very /\
.I\
very\
/
```

converts the line into four shorter lines, preceding the word "very" with the line ".I", and eliminating the spaces around the "very", all at the same time.

When a new line is substituted in a string, dot is left at the last line created.

5.4.9 Joining Lines

Lines may also be joined together, but this is done with the **j** command instead of **s**. Given the lines

```
Now is
 the time
```

suppose that dot is set to the first of them. Then the command

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a **j** command joins the lines signified by dot and dot + 1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example

```
1,$jp
```

joins all the lines into one big one and prints it.

5.4.10 Rearranging a Line - \(and \)

Recall that "&" is a shorthand for whatever was matched by the left side of an **s** command. In much the same way, you can capture separate pieces of what was matched. The only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose that you have a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in:

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone.

The alternative is to "tag" the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol, "\1", refers to whatever matched the first \(...\) pair; "\2", to the second \(...\), and so on.

The command

```
1,$s/^\([.*]\), *\(.*\)/\2 \1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with "\1". The second \(...\) is whatever follows the comma and any spaces, and is referred to as "\2".

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands g and v provide a way for you to print exactly those lines which were affected by the substitute command, and thus

verify that it did what you wanted in all cases.

5.5 Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains "thing". Then no address is required with commands like s to make a substitution on that line, or **p** to print it, or l to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there is no "thing"? Then you are left right where you were: dot is unchanged. This is also true if you were sitting on the only "thing" when you issued the command. The same rules hold for searches that use ?...?; the only difference is the direction in which you search.

The delete command, d, leaves dot pointing at the line that followed the last deleted line. When the line dollar ($) gets deleted, however, dot points at the new line "$".

The line-changing commands **a**, **c**, and **i**, by default, all affect the current line. If you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

The **a**, **c**, and **i** commands behave identically in one respect -- when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can type

```
a
text
botch    (minor error)
.
s/botch/correct/  (fix botched line)
a
more text
.
```

without specifying any line number for the substitute command or for the second append command. Or you can type:

```
a
text
horrible botch  (major error)
.
c       (replace entire line)
fixed up line
.
```

You should experiment to determine what happens if you add no lines with a, c, or i.

The r command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even type

```
0r
```

to read a file in at the beginning of the text. (You can also type 0a or 1i to start adding text at the beginning.)

The w command writes out the entire file. If you precede the command by one line number, that line is written. If you precede it by two line numbers, that range of lines is written. The w command does not change dot: the current line remains the same, regardless of what lines are written. This is true even if you type something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

(Since the w command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you accidentally delete what you're editing.)

The general rule is simple: you are left sitting on the last line changed; if there were no changes, then dot is

unchanged.

To illustrate, suppose that there are three lines in the buffer, and the line given by dot is the middle one:

```
x1
x2
x3
```

Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

5.5.1 Semicolon - ;

Searches with /.../ and ?...? start at the current line and move forward or backward, respectively, until they either find the pattern or get back to the current line. Sometimes this is not what you want. Suppose, for example, that the buffer contains lines like this:

```
   .
   .
   .
ab
   .
   .
   .
bc
   .
   .
   .
```

Starting at line 1, you would expect the command

```
/a/,/b/p
```

to print all the lines from the "ab" to the "bc" inclusive.

This is not what happens. Both searches (for "a" and for "b") start from the same point, and thus they both find the line that contains "ab". The result is to print a single line. Worse, if there had been a line with a "b" in it before the "ab" line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In ed, the semicolon ";" can be used just like comma, with the single difference that use of a semicolon forces dot to be set at the time the semicolon is encountered, as the line numbers are being evaluated. In effect, the semicolon "moves" dot. Thus, in our example above, the command

```
/a/;/b/p
```

prints the range of lines from "ab" to "bc", because after the "a" is found, dot is set to that line, and then "b" is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the second occurrence of "thing". You could type

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to type:

```
/thing/;//
```

This says to find the first occurrence of "thing", set dot to that line, then find the second and print only that.

Closely related is searching for the second to last occurrence of something, as in:

```
?something?;??
```

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to type

```
1;/thing/
```

because this fails if "thing" occurs on line 1. But it is possible to type

```
0;/thing/
```

because this starts the search at line 1. This is one of the few places where 0 is a legal line number.

5.5.2 Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you press the <INTERRUPT> key while ed is executing a command, your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable -- if you are reading or writing a file or making substitutions or deleting lines. These will be stopped in some unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus, if you decide to print until you see an interesting line, and then press <INTERRUPT>, dot will not not be set to that line or even near it. Dot is left where it was when the p command was started.

5.6 Cut and Paste with the Editor

Now we move on to manipulating pieces of files, individual lines or groups of lines. Time spent mastering these techniques is time well spent.

5.6.1 Inserting One File into Another

Suppose you have a file called memo, and you want the file called table to be inserted just after the reference to Table 1. That is, in memo somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in table has to go there, probably so it will be formatted properly by either of the text formatters nroff or troff. Now what?

This one is easy. Edit memo, find "Table 1", and add the file table right there:

```
ed memo
/Table 1/
Table 1 shows that ... response from ed
.r table
```

The critical line is the last one. The r command reads a file; here you asked for it to be read in right after line dot. An r command, without any address, adds lines at the end, so it is the same as "$r".

5.6.2 Writing Out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
[lots of stuff]
.TE
```

which is the way a table is set up for the tbl program. To isolate the table in a separate file called table, first find the start of the table (the ".TS" line), then write out the interesting part:

```
/^\.TS/
.TS [ed prints the line it found]
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the w command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. If you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it -- don't retype it. For example, in the editor, type:

```
a
lots of stuff
horrible line
.
.w  temp
a
more stuff
.
.r temp
a
more stuff
.
```

5.7 Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a "script," i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every "Xenix" to "XENIX" and every "USA" to "America" in a large number of files. Then put into the file script the lines:

```
g/Xenix/s//XENIX/g
g/USA/s//America/g
w
q
```

Now you can type:

```
ed filel <script
ed file2 <script
...
```

This causes ed to take its commands from the prepared script. Notice that the whole job has to be planned in advance, and that by using the XENIX shell command interpreter, you can cycle through a set of files automatically.

In preparing editing scripts, you will need to place a period as the only character on a line to indicate termination of input from an **a** or **i** command. This is difficult to do in ed, because the period you input will terminate input rather than be inserted. Nor will it do to escape the period with a backslash. One workable solution is to create the script using a character such as the at-sign (@) to indicate end of input. Then, later, use the

following command to replace the at-sign with a period:

```
s/^@$/./
```

This will replace the at-sign with the needed period.

5.8 Summary of Commands and Line Numbers

This following is a list of all ed commands and line numbers. The general form of ed commands is the command name, preceded by one or two optional line numbers and, in the case of e, f, r, and w, followed by a filename. Only one command is allowed per line, but a p command may follow any other command (except for e, f, r, w, and q).

a Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until a period is typed on a new line. The value of dot is set to the last line appended.

c Change the specified lines to the new text which follows. The new lines are terminated by a period on a new line, as with a. If no lines are specified, replace line dot. Dot is set to last line changed.

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

e Edit new file. Any previous contents of the buffer are thrown away, so issue a w before hand.

f Print remembered filename. If a name follows f, then the remembered name is set to it.

g The command g/string/commands executes commands on those lines that contain string, which can be any context search expression.

i Insert lines before specified line (or dot) until a single period is typed on a new line. Dot is set to the last line inserted.

l List lines, making visible non-printing ASCII characters and tabs. Otherwise similar to p.

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

p Print specified lines. If none are specified, print the line specified by dot. A single line number is equivalent to a line-numberp command. A single <RETURN> prints ".+1", the next line.

q Quit ed. Your work is not saved unless you first give a **w** command. Give it twice in a row to abort edit.

r Read a file into buffer (at end unless specified elsewhere.) Dot is set to the last line read.

s The command "s/string1/string2/" substitutes the pattern matched by string1 with the string specified by string2 in the specified lines. If no lines are specified, the substitution takes place only on the line specified by dot. Dot is set to the last line in which a substitution took place. which means that if no substitution takes place, dot remains unchanged. The s command changes only the first occurrence of string1 on a line; to change multiple occurrences on a line, type a **g** after the final slash.

t Transfer specified lines to the line named after t. Dot is set to the last line moved.

v The command v/string/commands executes commands on those lines that do not contain string.

u Undo the last substitute command.

w Write out the editing buffer to a file. Dot remains unchanged.

.= Print value of dot. (An equal sign by itself prints the value of $.)

! The line !command-line causes command-line to be executed as a XENIX command.

/string/
Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at .+1 , wraps around from $ to 1, and continues to dot, if necessary.

?string?
Context search in reverse direction. Start search at .-1 , scan to 1, wrap around to $.

CHAPTER 6

VI

CONTENTS

6.1 Introduction

Vi is a screen-oriented text editor that can be used for most any text editing purpose. It is well integrated into the XENIX environment and should be thought of, along with the shell, as one of the central XENIX tools. Vi offers a powerful set of text editing operations based on a mnemonic command set. Most commands are single keystrokes that perform simple editing functions. So that the terminal screen is used to the utmost, Vi displays a "window" into the file you are editing. This window can be changed quickly and easily within Vi, and provides visual feedback while editing (the name Vi itself is short for "visual").

As you use Vi, it is important to realize that it and the line editor Ex are one and the same editor: the names Vi and Ex identify a particular user interface rather than any underlying functional difference. The differences in user interface, however, are quite striking. Ex is a powerful line-oriented editor. However, visual updating of the terminal screen is limited, and commands are entered on a command line. Vi, on the other hand, is a screen-oriented editor designed so that what you see on the screen corresponds exactly and immediately with the contents of the file you are editing. Most Vi commands are single keystroke mnemonics that do not require a command line.

For most editing purposes, you will want to use Vi rather than either of the editors Ed or Ex because of the superior way in which Vi displays file contents on the screen. It is important, however, to realize that many of the commands available to you in Ex also work in Vi. Those commands that work in an identical fashion are those that you will type on the bottom status line in Vi. Keep this Vi/Ex split in mind as you use the editor -- it will help to eliminate confusion that can arise when learning the editor.

6.2 Demonstration Run

The following demonstration run gives you hands on experience using Vi. It should give you some initial satisfaction that you can use it for most any editing purpose. Remember that the best way to learn Vi is to actually use it, so don't be afraid to experiment.

NOTE: Most of the Vi commands in this demonstration run are single key strokes that do not require a terminating <RETURN>. Therefore, do not assume that a <RETURN> is needed after the entry of each command. Commands that do require a terminating <RETURN> are called **Ex** commands and are entered as command lines on the Vi status line.

To begin, you must first make sure that your terminal has been properly set up. Most terminals are supported, so there should be no problem. However, this demonstration run presumes that Vi knows about the terminal you are using. See Section 6.8 "Start-Up Files and Options," for more information about setting up your terminal for use with Vi.

To enter the editor type:

```
vi temp
```

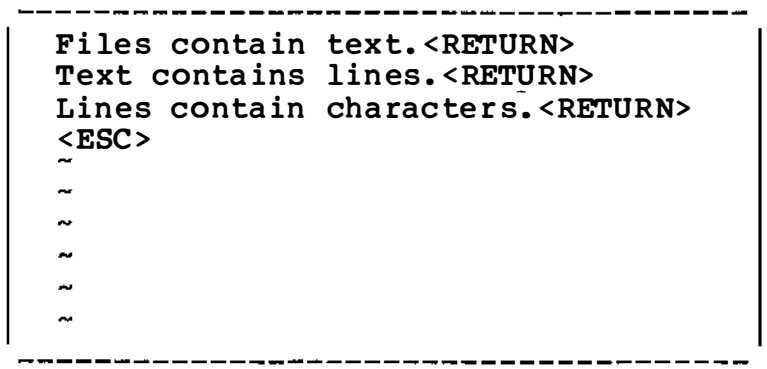
This invokes the editor and places you in Vi command mode, where the keys you press are interpreted as editing commands. The editor then clears your screen and prints out a row of tildes (~). You are initially editing a temporary file called the editing buffer. The top line of your display is the only line in the editing buffer and is marked by the cursor. The line containing the cursor is called the "current line." The lines containing tildes are not part of the editing buffer: they indicate lines on the screen only, not real lines in the editing buffer. When you write out the editing buffer, you will write to the file named temp, which is the same as the file you named when you invoked the editor. In our examples, the cursor will be indicated by an underscore, as shown below:

Note that we show a shrunken ten line screen to save space. In reality, however, Vi takes advantage of whatever size screen you have.

To begin, create some text by using the **i** (insert) command. To do this, type:

```
i
```

Next, type the following three lines of text to give you something to play around with (note that <RETURN> and <ESC> are single keys):



Like most commands, the **i** command is mnemonic (for insert) and is not echoed on your screen. The command itself switches you from command mode to insert mode. Once in insert mode, the characters you type are inserted into the editing buffer; they are <u>not</u> interpreted as Vi commands. To exit insert mode and reenter command mode you will always need to type <ESC>. This switching between modes occurs often in Vi, so get used to it now.

Next comes a command that you'll use frequently in Vi: the repeat command. The repeat command repeats the most recent insert or delete command. Since we have just executed an

insert command, the repeat command will repeat the insertion, duplicating the inserted text. The repeat command is executed by typing a period (.) or "dot". So, just type

to insert another three lines of text. The command is repeated relative to the location of the cursor and inserts text below the current line. (The current line is always the line containing the cursor.) After you type dot, your screen will look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Files contain text.
Text contains lines.
Lines contain characters.
~
~
~
```

Another command which is very useful (and which you'll need often in the beginning) is the undo command, u. Type

u

and notice that the three lines you just finished inserting are deleted or "undone":

```
Files contain text.
Text contains lines.
Lines contain characters.
~
~
~
~
~
~
```

Now type

u

again, and the three lines are reinserted! This undo feature can be remarkably useful in recovering from inadvertent deletions or insertions. Notice that in contrast to the repeat command (.), the undo command inverts the last insert or delete command. And because the undo command is an insert or delete command itself, two consecutive undo commands cancel each other out.

Now lets learn how to move the cursor around in Vi. Typing the keys "h", "j", "k", and "l", will move the cursor left, down, up, and right, respectively. Note that these keys are chosen because of their relative positions on the keyboard, not for any mnemonic reason. On most terminals, you can also use the arrow keys to move in the same way. Remember that the "h", "j", "k", and "l" keys and the arrow keys only work when in command mode.

Try moving the cursor using these keys. When you are done, type the **H** command to home the cursor in the upper left corner of the screen. Then type the **L** command to move to the lowest line on the screen. (Note that case is significant in our example: **L** moves to the lowest line on the screen; while **l** will move the cursor forward one character.) Next, try moving the cursor to the last line in the file with the "goto" command, **G**. If you type "2G", the cursor will move to the beginning of the second line in the file; if you have a 10,000 line file, and type "8888G", the cursor will go to the beginning of line 8888.

The above cursor movement commands should allow you to move around well enough for this demonstration run. Other cursor movement commands you might want to try out are: **w**, to move forward a word; **b**, to back up a word; **0** to move to the beginning of a line; and **$** to move to the end of a line.

Several screen-oriented scrolling commands also exist. These are all mnemonically named control characters:

<CONTROL-U> Scroll up

<CONTROL-D> Scroll down

<CONTROL-F> Page forward one screenful

<CONTROL-B> Page backwards one screenful

You can also search forward for a string of characters by typing a slash (/) followed by the string of characters you are searching for, terminated by a <RETURN>. For example, type

H

to move the cursor to the top of the screen, then type

/char<RETURN>

as shown below:

```
+-------------------------------------+
|  Files contain text.                |
|  Text contains lines.               |
|  Lines contain characters.          |
|  Files contain text.                |
|  Text contains lines.               |
|  Lines contain characters.          |
|  ~                                  |
|  ~                                  |
|  ~                                  |
|  ~                                  |
|  /char<RETURN>                      |
+-------------------------------------+
```

Your cursor moves to the beginning of the word "characters" on line three. To search for the next occurrence of the string "char', simply type the character "n". This will take you to the beginning of the word "characters" on line six. If you type "n" again, Vi will search past the end of the file, wrap around to the beginning, and again find the occurrence of the string "char" on line three.

Note that the slash character and the string that you were searching for appear at the bottom of the screen before you type. This bottom line is the Vi status line. It is used to display several different kinds of information:

1. Ex commands
2. Strings that you are searching for
3. Buffer status information
4. Error messages
5. Any other information that needs to be distinguished from the text that is part of your file

For example, to get status information about the editing buffer, type <CONTROL-G>. This tells you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and the percentage of the file (in lines) that precedes the

cursor (i.e., where you are in the file relative to the beginning). All this information is given on the status line.

Now that we know how to insert and create text, and how to move around within the editing buffer, we're ready to delete text. The three most common delete commands are:

dd Delete the current line (i.e., the line in which the cursor resides).

dw Delete the word in which the cursor resides.

x Delete the character beneath the cursor.

D Delete from the cursor to the end of the line.

d0 Delete from the cursor to the start of the line.

. Repeat the last change. (Use this only if your last command was a deletion and not an insertion, otherwise you'll insert text instead of delete it.)

To learn how all these commands work, we'll step you through the deletion of various parts of the editing buffer. To begin, move to the first line of your editing buffer by typing:

```
1G
```

At first, your editing buffer should look like this:

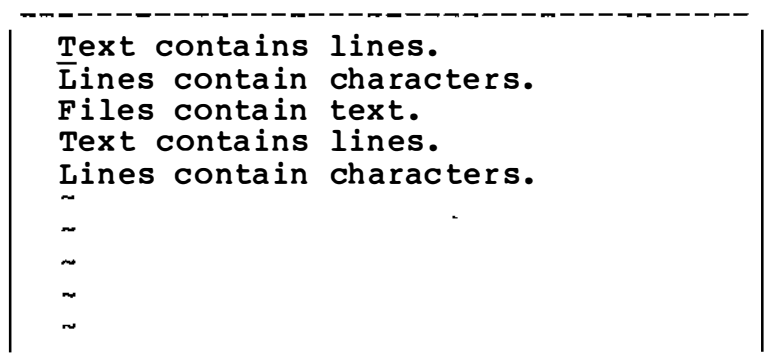
```
Files contain text.
Text contains lines.
Lines contain characters.
Files contain text.
Text contains lines.
Lines contain characters.
~
~
~
```

Now type

```
dd
```

to delete the first line. Your editing buffer now looks

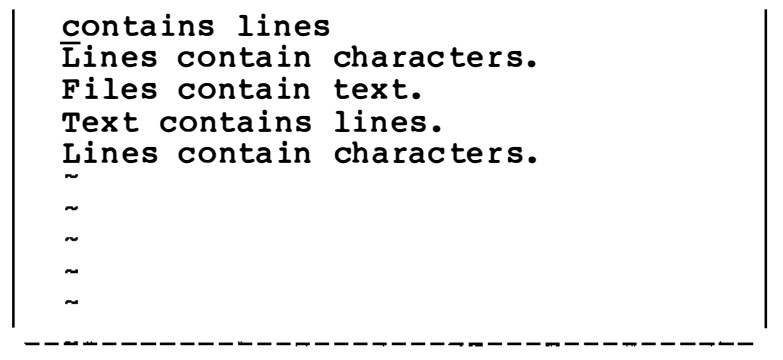
like this:



Next type

```
dw
```

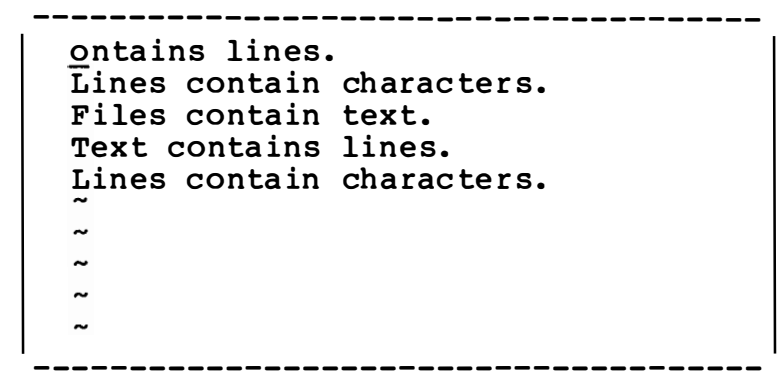
to delete the word in which the cursor resides. Your file now looks like this:



You can quickly delete the character beneath the cursor by typing:

```
x
```

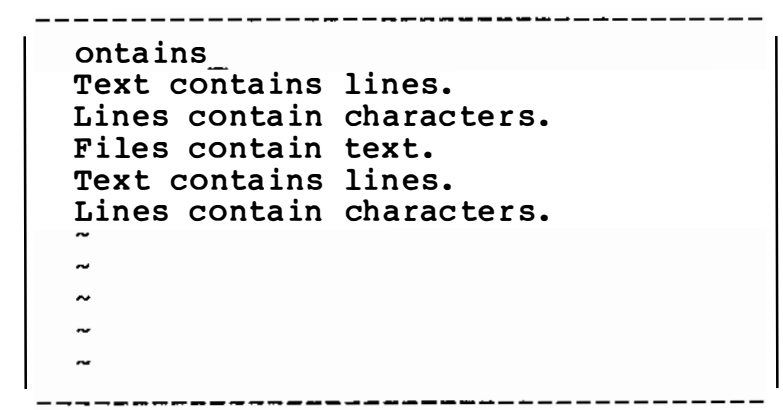
This leaves:



Now type a "w" command to move your cursor to the beginning of the word "lines" on line one. Then type an uppercase "D" to delete to the end of the line:

```
D
```

This leaves your editing buffer looking like this:



Now type

```
d0
```

to delete all characters on the line before the cursor. This leaves a single space on the line:

```
Lines contain characters.
Files contain text.
Text contains lines.
Lines contain characters.
~
~
~
~
~
```

All of the editing you have been doing has affected the editing buffer, and not the file named temp that you specified when you invoked Vi. To write the editing buffer out to temp, use the Ex write command:

```
Lines contain characters.
Files contain text.
Text contains lines.
Lines contain characters.
~
~
~
~
:w<RETURN>
```

All Ex commands are preceded by a colon which acts as a prompt on the status line. Ex commands themselves are entered on this line and terminated with a <RETURN>. If you want to write the editing buffer out to a file other than temp, you can give the w command a filename argument that specifies the name of the file you want to write to.

In general, Ex commands allow you to interface with the operating system. For instance, you can read in the contents of a file below the current line by typing

```
:r filename<RETURN>
```

where filename is the name of the file you want to read in. You can also execute arbitrary XENIX commands such as date, by typing:

```
:!date<RETURN>
```

This will output the date and then prompt you to press

<RETURN> to reenter Vi command mode. Go ahead and try it.

Note that when you execute Ex commands, you are really executing commands available in the line-oriented editor called Ex. Ex and Vi are really one and the same editor, the only difference between the two is the user interface.

Besides the set of editing commands described above, there are a number of options that can be set either when you invoke Vi, or later when editing. These options allow you to control a large number of editing parameters. For example, you can specify automatic line numbering, automatic word wrap, and whether or not case is significant in string searches. You can get a complete list of the available options by typing:

```
:set all<RETURN>
```

How to set these options is described in Section 6.7, "Start-Up Files and Options," but it is important now that you be aware of their existence. Depending on what you are doing, and your own personal preferences, you will want to alter the default settings for many of these options.

Finally, to exit Vi and save the editing buffer to the file named temp, type:

```
:x<RETURN>
```

If you have made any changes to the editing buffer, this writes out the editing buffer to the last named file and then exits the editor. If you don't want to save the editing buffer, you can abort the editing session by typing:

```
:q!<RETURN>
```

This completes the demonstration run. There are still many commands that have not been shown you, but nevertheless, the fundamentals of using Vi have been covered. You should now know how to get into and out of Vi, how to insert and delete text, how to move your cursor around, how to execute Ex commands, and how to write out the editing buffer to a file. Following sections give you more detailed information about the commands covered above and about Vi's other commands and features.

6.3 Basic Concepts

To use Vi effectively, you will need to understand the basic concepts that are essential in understanding how Vi works. The topics discussed here include:

- The Editing Buffer
- Modes of Operation
- Special Keys
- Text Objects

6.3.1 The Editing Buffer

Vi performs no editing operations on the file that you name during invocation. Instead, it works on a copy of the file in an editing buffer. The editor remembers the name of the file specified at invocation, so that it can later copy the editing buffer back to the named file. This means that you do not affect the contents of the named file unless and until you explicitly copy the changes you have made back to the original file. This setup allows you to edit the buffer without immediately destroying the contents of the original file. See Figure 6-1 for an illustration of how this all works.

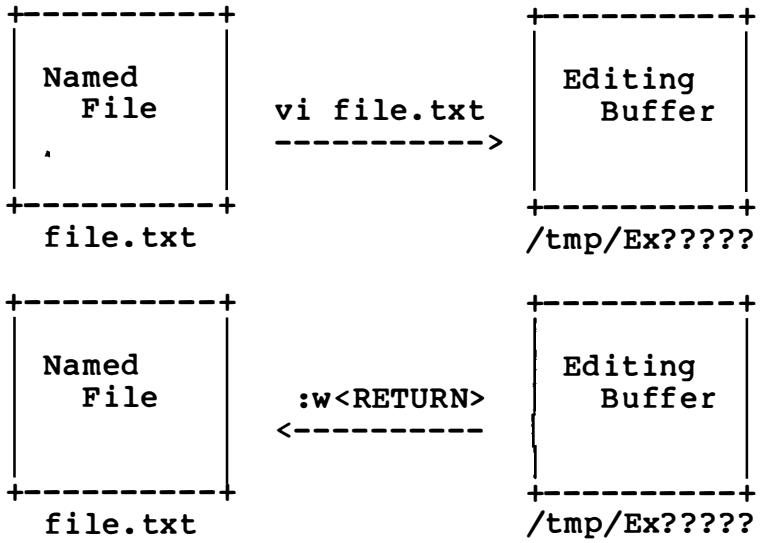


Figure 6-1. The Editing Buffer

When you invoke Vi with a single filename argument, the named file is copied to a temporary editing buffer. When the file is written out, the temporary file is written back to the named file.

6.3.2 Modes of Operation

Before using Vi extensively you need to clearly understand the concept of mode. Within Vi there are three distinct modes of operations:

Command Mode

After invoking Vi, you are automatically placed in Vi command mode. Within command mode, the keys that you press are interpreted as editing commands. In most cases, the keys that you press are not echoed on the screen. Because they are not echoed, you get no visual feedback on the command you are typing. This isn't bad for single keystroke commands, but it sometimes can be confusing for commands that require several keystrokes. However, if you ever get confused when typing a command, you can abort command entry by pressing <INTERRUPT>.

Insert Mode

Insert mode can be entered by typing any of the Vi insert, append, open, substitute, change, or replace commands. A few other commands also put

you into insert mode. Once in insert mode, the keys you type are inserted into your editing buffer as you type them. Some characters are special in insert mode; these are listed and described below:

<BKSP>
This backs up the cursor one character on the current line. The last character typed before the <BKSP> is removed from the input buffer, but remains displayed on the screen.

<CONTROL-U>
Moves the cursor back to the first character of the insertion, and restarts insertion.

<CONTROL-V>
Removes the special significance of the next typed character. Use <CONTROL-V> to insert control characters. Note that line feed <LF> and <CONTROL-J> cannot be inserted in the text except as newline characters. Also, <CONTROL-Q> and <CONTROL-S> are trapped by the operating system before they are interpreted by Vi, so they too cannot be inserted.

<CONTROL-W>
Moves the cursor back to the first character of the last inserted word.

<CONTROL-D>
Backtabs over whitespace at the beginning of a line. Otherwise, if the autoindent option is set this whitespace cannot be backspaced over.

<CONTROL-T>
During an insertion, with the autoindent option set and at the beginning of the current line, typing this character will insert shiftwidth white space.

<CONTROL-@>
If typed as the first character of an insertion it is replaced with the last text inserted, and the insertion terminates. Only 128 characters are saved from the last insertion. If more than 128 characters were inserted, then this command inserts no characters. A <CONTROL-@> cannot be part of

a file, even if quoted.

Ex Escape Mode

The Vi and Ex editors are one and the same editor differing mainly in their user interface. In Vi, as we have seen, commands are usually single keystrokes. In Ex, commands are lines of text terminated by a <RETURN>. Vi has a special "escape" command that gives you access to many of theses line-oriented Ex commands. To escape to Ex escape mode, you need only type a colon (:). The colon is echoed on the bottom status line as a prompt for the Ex command that you want to execute. You then may type in the command, followed by a <RETURN> or an <ESC>. An executing command can also be aborted by typing <INTERRUPT>. Most file manipulation commands are executed in Ex escape mode; for example, the commands to read in a file, and to write out the editing buffer to a file. Other Ex commands that you need to know are the commands to perform global substitutions and to quit the editor. For more information, see Section 6.6 "Ex Commands."

6.3.3 **Special Keys**

There are several keys that you'll use over and over when editing in Vi. These keys keys are used to edit, delimit, or abort commands and command lines. The first key that we'll look at is the <ESC> key. It should be near the upper left corner of your terminal. Try pressing it a few times -- it rings the bell to indicate that Vi is in its normal command state. On some terminals, the editor quietly flashes the screen rather than ringing the bell. In general, the <ESC> key always returns you to Vi command mode. You'll use it most often in exiting from insert mode. It can also be used to terminate Ex command lines. In addition, partially formed commands are canceled by typing an <ESC>. This key is a fairly harmless one to press, so you should press it when you don't know exactly what is going on.

The <RETURN> key is used to terminate Ex commands when in Ex escape mode. You also type <RETURN> whenever you want to start a new line when in insert mode.

Another useful key is the <INTERRUPT> key, which is often the same as the <DEL> or <RUBOUT> key on many terminals. The <INTERRUPT> key, as its name implies, generates an interrupt, telling the editor to stop what it is doing. You can use <INTERRUPT> to abort any command that is executing.

It is a forceful way of making the editor listen to you, or to return to Vi command mode, if you don't know or don't like what is going on.

The next key of interest is the slash (/) key. This key is used when you want to specify a string to be searched for. When you type it, the slash appears on the bottom status line as a prompt for a search string. You can then enter the search string, followed by a <RETURN> or an <ESC>. You can get the cursor back to the current position by pressing the <INTERRUPT> key. Note that the question mark (?) works exactly like the slash key, except that it is used to search backwards in a file instead of forwards.

The last key that we'll discuss is the colon (:). When you type a colon, it is echoed on the status line as a prompt for an Ex command. You can then type in any Ex command, followed by an <ESC> or <RETURN> and the given Ex command will be executed.

6.3.4 Text Objects

The editing operations of the Vi editor are in most cases based on the notion of a text object. A text object is any sequence of consecutive characters in a line or any sequence of consecutive lines in a file. In general, text objects are delimited by the cursor and a cursor movement command. The character on which the cursor sits delimits one end of an intra-line object such as a word; the current line delimits one end of a multi-line object such as a line number range. Because the location of the cursor is always known by Vi, text objects are normally specified by simply typing the appropriate cursor movement command. Thus, naked cursor movement commands move the cursor to the opposite end of a given text object. Text objects can be moved, changed, or deleted by combining the appropriate command operator with a cursor movement command.

6.4 Invoking and Exiting Vi

The Vi invocation syntax is as follows:

vi [-option ...] [+command] [filename ...]

The simplest form of this syntax, and the easiest way to enter Vi is to type:

vi

This allows you to work on an empty editing buffer. The most common way to enter Vi, however, is to specify one or more filenames as shown below:

vi filename ...

Filename arguments indicate files to be edited. You can also enter the editor and then move to a particular place in a file by giving Vi a single line number or search string argument, preceded by a plus (+).

Examples:

vi	Edit empty editing buffer
vi file	Edit named file
vi +123 file	Goto line 123
vi +45 file	Goto line 45
vi +/dog file	Find first occurrence of "dog"
vi +/tty file	Find first occurrence of "tty"

Vi may be invoked with any of the following options:

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

-r This 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, printing a list of saved files.

-x This option causes Vi to prompt for a key used to encrypt and decrypt the contents of the named files.

-R This sets a readonly option so that files can be viewed but not edited.

There are several ways to exit the editor. One way is to type:

:wq<RETURN>

This command writes the editing buffer to the file you are editing, quits the editor, and then returns you to the XENIX shell. Similarly, if you type

ZZ

the same thing happens, except that the editing buffer is written to the file you are editing only if you have made

any changes. The "ZZ" command is equivalent to the command:

:x<RETURN>

To abort an editing session, type

:q!<RETURN>

The exclamation mark (!) tells Vi to quit unconditionally. In this case, the editing buffer is not written out. If you type just

:q<RETURN>

the editor will not let you quit unless you have written out your file and will print the message:

No write since last change (":q!" overrides)

This tells you to use ":q!" if you really want to quit without writing out your file.

6.5 Vi Commands

Vi is a visual editor with a window on the file. What you see on the screen is Vi's notion of what the file contains. Most commands do not cause any change to the screen until the complete command is typed. Also, most commands may take a preceding count that specifies repetition of the command. This count parameter is not given in the syntax, but is implied unless overriden by some other prefix argument.

Should you get confused while typing a command, you can abort the command by typing an <INTERRUPT> character. Usually typing an <ESC> will produce the same result. When Vi gets an improperly formatted command it also rings a bell.

Following subsections describe Vi commands. These are not all of the commands available in Vi. By typing a colon, you can enter Ex command mode and enter Ex commands. For more information, see Section 6.6, "Ex Commands"

6.5.1 Cursor Movement

The cursor movement keys allow you to move your cursor around in a file. Note in particular the arrow keys (if available on your terminal), the "hjkl" cursor keys, and <SPACE>, <BKSP>, <CONTROL-N>, and <CONTROL-P>. These three sets of keys perform identical functions; the choice of which to use is entirely up to you.

Forward Space - l, <SPACE>, or -->

Syntax: **l**
<SPACE>
-->

Function: Move cursor forward one character. If a count is given, move forward count characters. Note that you cannot move past the end of the line.

Backspace - h, <BKSP>, or <--

Syntax: h
<BKSP>
<--

Function: Move cursor backward one character. If a count is given, move backwards count characters. Note that you cannot move past the beginning of the current line.

Next Line - +, <RETURN>, j, <CONTROL-N>, and <LF>

Syntax: **+**
<RETURN>

Function: Move cursor down to the beginning of the next line.

Syntax: **j**
<CONTROL-N>
<LF>
(down arrow)

Function: Move cursor down one line, remaining in the same column. Note the difference between these commands and the preceding set of next line commands which move to the beginning of the next line.

Previous Line - k, <CONTROL-P>, and -

Syntax: **k**
<CONTROL-P>
(up arrow)

Function: Move cursor up one line, remaining in the same column. If a count is given then the cursor is moved up a number of lines equal to the count.

Syntax: **-**

Function: Move cursor up to the beginning of the previous line. If a count is given then the cursor is moved up a number of lines equal to the count.

Beginning of Line - 0 and ^

Syntax: **^**
0

Function: Move cursor to the beginning of the current line. Note that **0** always moves the cursor to the first character of the current line. The circumflex (^) works somewhat differently: it moves to the first character on a line that is not a tab or a space. This is useful when editing files that have a great deal of indentation, such as program texts.

End of Line - $

Syntax: $

Function: Move cursor to the end of the current line. Note that the cursor resides on top of the last character on the line. If a count is given, then the cursor is moved forward count-1 lines to the end of the line.

Goto Line - G

Syntax: [linenumber]**G**

Function: Go to the beginning of the line specified by linenumber. If no linenumber is given, the cursor moves to the beginning of the last line in the file. To find the line number of the

current line, use <CONTROL-G>. To turn on automatic line numbering on your screen, type:

```
:set linenumber<RETURN>
```

Column - |

Syntax: [column]|

Function: Move cursor to the column in the current line given by column. If no column is given then the cursor is moved to the first column in the current line.

Word Forward - w and W

Syntax: w
W

Function: Move cursor forward to the beginning of the next word. The lowercase **w** command searches for a word defined as a string of alphanumeric characters separated by punctuation or whitespace (i.e., tab, newline, or space characters). The uppercase **W** command searches for a word defined as a string of non-whitespace characters.

Back Word - b and B

Syntax: b
B

Function: Move cursor backward to the beginning of a word. The lowercase b command searches backwards for a word defined as a string of alphanumeric characters separated by punctuation or whitespace (i.e., tab, newline, or space characters). The uppercase **B** command searches for a word defined as a string of non-whitespace characters. If the cursor is already within a word, then it moves backwards to the beginning of that word.

End - e and E

Syntax: **e**
E

Function: Move cursor to the end of a word. The lowercase e command moves the cursor to the last character of a word, where a word is defined as a string of alphanumeric characters separated by punctuation or whitespace (i.e., tab, newline, or space characters). The uppercase **E** moves the cursor to the last character of a word where a word is defined as a string of non-whitespace characters. If the cursor is already within a word, then it moves to the end of that word.

Sentence - (and)

Syntax: **(**
)

Function: Move cursor to the beginning (left parenthesis) or end of a sentence (right parenthesis). A sentence is defined as a sequence of characters ending with a period (.), question mark (?), or exclamation mark (!), followed by either two spaces or a newline. A sentence begins on the first non-whitespace character following a preceding sentence. Sentences are also delimited by paragraph and section delimiters. See below.

Paragraph - { and }

Syntax: **}**
{

Function: Move cursor to the beginning ({) or end (}) of a paragraph. A paragraph is defined with the paragraphs option. By default, paragraphs are delimited by the nroff macros ".IP", ".LP", ".PP", ".QP", and ".bp". Paragraphs also begin after empty lines.

Section - [[and]]

Syntax: **]]**
[[

Function: Move cursor to the beginning ([[) or end (]]) of a section. A section is defined with the sections option. By default, sections are delimited by the nroff macros ".NH" and ".SH". Sections also start at formfeeds (<CONTROL-L>) and at lines beginning with a brace ({).

Match Delimiter - %

Syntax: **%**

Function: Move cursor to a matching delimiter, where a delimiter is a parenthesis, a bracket, or a brace. This is useful when matching pairs of nested parentheses, brackets, and braces.

Home - H

Syntax: [offset]**H**

Function: Home cursor to upper left corner of screen. Use this command to quickly move to the top of the screen. If an offset is given, then the cursor is homed offset-1 number of lines from the top of the screen. Note that the command "dH" deletes all lines from the current line to the top line shown on the screen.

Middle Screen - M

Syntax: **M**

Function: Move cursor to the beginning of the screen's middle line. Use this command to quickly move to the middle of the screen from either the top or the bottom. Note that the command "dM" deletes from the current line to the line specified by the **M** command.

Lower Screen - L

Syntax: [offset]L

Function: Move cursor to the lowest line on the screen. Use this command to quickly move to the bottom of the screen. If an offset is given, then the cursor is homed offset-1 number of lines from the bottom of the screen. Note that the command "dL" deletes all lines from the current line to the bottom line shown on the screen.

Previous Context - `` and ''

Syntax: ''
'character
``
`character

Function: Move cursor to previous context or to context marked with the m command. If the single quote or back quote is doubled, then the cursor is moved to previous context. If a single character is given after either quote, then the cursor is moved to the location of the specified mark as defined by the m command. Previous context is the location in the file of the last "non-relative" cursor movement. The single quote (') syntax is used to move to the beginning of the line representing the previous context. The back quote (`) syntax is used to move to the previous context within a line.

6.5.2 The Screen Commands

The screen commands are not cursor movement commands and cannot be used in delete commands as the delimiters of text objects. However, the screen commands do move the cursor and are useful in paging or scrolling through a file. These commands are described below:

Page - <CONTROL-U> and <CONTROL-D>

Syntax: [size]<CONTROL-U>
[size]<CONTROL-D>

Function: Scroll screen up a half window (<CONTROL-U>) or down a half window (<CONTROL-D>). If size is

given, then the scroll is size number of lines. This value is remembered for all later scrolling commands.

Scroll - <CONTROL-F> and <CONTROL-B>

Syntax: **<CONTROL-F>**
<CONTROL-B>

Function: Page screen forward and backwards. Two lines of continuity are kept between pages if possible. A preceding count gives the number of pages to move forward or backwards.

Status - <CONTROL-G>

Syntax: **<BELL>**
<CONTROL-G>

Function: Print Vi status on status line. This gives you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and the percentage of the file (in lines) that precedes the cursor.

Zero Screen - z

Syntax: [linenumber]**z**[size]**<RETURN>**
[linenumber]**z**[size]**.**
[linenumber]**z**[size]**-**

Function: Redraw the display with the current line zeroed at the top, bottom, or middle of the screen. If you give a size, then the number of lines displayed is equal to size. If a preceding linenumber is given, then the given line is zeroed at the top of the screen. If the last argument is a <RETURN>, then the current line is zeroed at the top of the screen. If the last argument is a period (.), then the current line is zeroed in the middle of the screen. If the last argument is a minus sign (-), then the current line is zeroed at the bottom of the screen.

Redraw - <CONTROL-R> or <CONTROL-L>

Syntax: **<CONTROL-R>**
<CONTROL-L>

Function: Redraw the screen. Use this command to erase any system messages that may scramble your screen. These messages do not affect the file you are editing.

6.5.3 Text Insertion

The text insertion commands always place you in insert mode. Exit from insert mode is always done by pressing <ESC>. The following insertion commands are "pure" insertion commands, no text is deleted when you use them. This differs from the text modification commands -- change, replace, and substitute -- which delete and then insert text in one operation.

Insert - i and I

Syntax: **i**[text]**<ESC>**
I[text]**<ESC>**

Function: Insert text in editing buffer. The lowercase **i** command places you in insert mode. Text is inserted before the character beneath the cursor. To insert a newline, just press a <RETURN>. Exit insert mode by typing the <ESC> key. The uppercase **I** command places you in insert mode, but begins text insertion at the beginning of the current line, rather than before the cursor.

Append - a and A

Syntax: **a**[text]**<ESC>**
A[text]**<ESC>**

Function: Append text to editing buffer. The lowercase **a** command works exactly like the lowercase **i** command, except that text insertion begins after the cursor and not before. This is the only way to add text to the end of a line. The uppercase **A** command begins appending text at the end of the current line rather than after the cursor.

Open New Line - o and O

Syntax: o[text]<ESC>
O[text]<ESC>

Function: Open a new line and insert text. The lowercase **o** command opens a new line below the current line; uppercase **O** opens a new line above the current line. After the new line has been opened, both these commands work like the **I** command.

6.5.4 Text Deletion

Many of the text deletion commands use the letter "d" as an operator. This operator deletes text objects delimited by the cursor and a cursor movement command. Deleted text is always saved away in a buffer. The delete commands are described below:

Delete Character - x and X

Syntax: **x**
X

Function: Delete a character. The lowercase **x** command deletes the character beneath the cursor. With a preceding count, cnt characters are deleted to the left beginning with the character beneath the cursor. This is the quick and easy way to delete a few characters. The uppercase X command deletes the character just before the cursor. With a preceding count, cnt characters are deleted backwards, beginning with the character just before the cursor.

Delete - d and D

Syntax: dcursor-movement
dd
D

Function: Delete text object. The lowercase **d** command takes a cursor-movement as an argument. If the cursor-movement is an intra-line command, then deletion takes place from the cursor to the end of the text object delimited by the cursor-movement. Deletion forward deletes the

character beneath the cursor; deletion backwards does not. If the cursor-movement is a multi-line command, then deletion takes place from and including the current line to the text object delimited by the cursor-movement.

The **dd** command deletes whole lines. The uppercase **D** command deletes from and including the cursor to the end of the current line.

Deleted text is automatically pushed on a stack of buffers numbered 1 through 9. The most recently deleted text is placed in a special delete buffer that is logically buffer 0. This special buffer is the default buffer for all delete, put, and yank commands. The buffers 1 through 9 can be accessed with the **p** and **P** ("put") commands using the double quotation mark (") to specify the number of the buffer. For example

```
"4p
```

puts the contents of delete buffer number 4 in your editing buffer just below the current line. Note that the last deleted text is "put" by default and does not need a preceding buffer number.

6.5.5 Text Modification

The text modification commands all involve the replacement of text with other text. This means that some text will necessarily be deleted. All text modification commands can be "undone" with the **u** command, discussed below:

Undo - u and U

Syntax: u
U

Function: Undo the last insert or delete command. The lowercase u command undoes the last insert or delete command. This means that after an insert, **u** deletes text; and after a delete, **u** inserts text. For the purposes of undo, all text modification commands are considered insertions.

The uppercase **U** command restores the current line to its state before it was edited, no matter how many times the current line has been edited since you moved to it.

Repeat - .

Syntax: .

Function: Repeat the last insert or delete command. A special case exists for repeating the **p** and **P** "put" commands. When these commands are preceded by the name of a delete buffer, then successive **u** commands print out the contents of the delete buffers.

Change - c and C

Syntax: c*cursor-movement* *text*<ESC>
C*text*<ESC>
cc*text*<ESC>

Function: Change a text object and replace it with *text*. Text is inserted as with the **i** command. A dollar sign ($) marks the extent of the change. The **c** command changes arbitrary text objects delimited by the cursor and a *cursor-movement*. The **C** and cc commands affect whole lines and are identical in function.

Replace - r and R

Syntax: r*char*
R*text*<ESC>

Function: Overstrike character or line with *char* or *text*, respectively. Use r to overstrike single characters and **R** to overstrike whole lines.

Substitute - s and S

Syntax: s*text*<ESC>
S*text*<ESC>

Function: Substitute current character or current line with *text*. Use s to replace a single character with new text. Use **S** to replace the current

line with new text. If a preceding count is given, then text substitutes for count number of characters or lines depending on whether the command is s or **S**, respectively.

Filter - !

Syntax: !cursor-movement cmd<RETURN>

Function: Filter the text object delimited by the cursor and the cursor-movement through the XENIX command, cmd. For example, the following command sorts all lines between the cursor and the bottom of the screen, substituting the designated lines with the sorted lines:

```
!Lsort
```

Join Lines - J

Syntax: **J**

Function: Join the current line with the following line. If a count is given, then count lines are joined.

Shift - < and >

Syntax:
```
>[cursor-movement]
<[cursor-movement]
>>
<<
```

Function: Shift text left (>) or right (<). Text is shifted by the value of the option shiftwidth, which is normally set to eight spaces. Both the > and < commands shift all lines in the text object delimited by the current line and cursor-movement. The >> and << commands affect whole lines. All versions of the command can take a preceding count that acts to multiply the number of objects affected.

6.5.6 Text Movement

The text movement commands move text in and out of the named buffers "a"-"z" and out of the delete buffers 1-9. These commands either "yank" text out of the editing buffer and into a named buffer or "put" text into the editing buffer from a named buffer or a delete buffer. By default, text is put and yanked from the "unnamed buffer", which is also where the most recently deleted text is placed. Thus it is quite reasonable to delete text, move your cursor to the location where you want the deleted text placed, and then put the text back into the editing buffer at this new location with the p or P command.

The named buffers are most useful for keeping track of several chunks of text that you want to keep on hand for later access, movement, or rearrangement. These buffers are named with the letters "a" through "z". To refer to one of these buffers (or one of the numbered delete buffers) in a command such as a put, yank, or delete command, use a quotation mark. For example, to yank a line into the buffer named a, type:

```
"ayy
```

To put this text back into the file, type:

```
"ap
```

If you delete text into the buffer named A rather than a, then text is appended to the buffer.

Note that the contents of the named buffers are not destroyed when you switch files. Therefore, you can delete or yank text into a buffer, switch files, and then do a put. Beware, buffer contents are destroyed when you exit the editor, so be careful.

Put - p and P

Syntax: ["alphanumeric]p
["alphanumeric]P

Function: Put text from a buffer into the editing buffer. If no buffer name is specified, then text is put from the unnamed buffer. The lowercase p command puts text either below the current line or after the cursor, depending on whether the buffer contains a partial line or not. The uppercase P command puts text either above the

current line or before the cursor, again depending on whether the buffer contains a partial line or not.

Yank - y and Y

Syntax: ["letter]ycursor-movement
["letter]yy
["letter]Y

Function: Copy text in the editing buffer to a named buffer. If no buffer name is specified, then text is yanked into the unnamed buffer. If an uppercase letter is used, then text is appended to the buffer and does not overwrite and destroy the previous contents.

The **Y** and **yy** commands yank lines.

6.5.7 Searching

The search commands search either forward or backwards in the editing buffer for a regular expression. For more information about regular expressions, see Section 6.8 "Regular Expressions".

Search - / and ?

Syntax: /[string]/<RETURN>
?[string]?<RETURN>

Function: Search forward (/) or backward (?) for string. A string is actually a regular expression as defined in Section 6.8 "Regular Expressions." The trailing delimiter is not required. See also the ignorecase and magic options.

Next String - n and N

Syntax: n
N

Function: Repeat the last search command. The n command repeats the search in the same direction as the last search command. The **N** command repeats the search in the opposite direction of the last search command.

Find Character - f and F

Syntax: fchar
Fchar
;
,

Function: Find character on current line. The lowercase f searches forward on the line; the uppercase F searches backwards. The semi-colon (;) repeats the last character search. The comma, (,) reverses the direction of the search.

To Character - t and T

Syntax: tchar
Tchar
;
,

Function: Move cursor up to but not on char. The semi-colon (;) repeats the last character search. The comma, (,) reverses the direction of the search.

Mark - m

Syntax: mletter

Function: Mark place in file with a lowercase letter. You can move to a mark using the "to mark" commands described below. It is often useful to create a mark, move the cursor, and then delete from the cursor to the mark with the following command: "d'letter".

To Mark - ' and `

Syntax: 'letter
`letter

Function: Move to letter. These commands let you move to the location of a mark. Marks are denoted by single lowercase alphabetic characters. Before you can move to a mark, it must first have been created with the m command. The back quote (') moves you to the exact location of the mark within a line; the forward quote (') moves you

to the beginning of the line containing the mark. Note that these commands are also legal cursor movement commands.

6.5.8 Exit and Escape Commands

There are several commands that are used to escape from Vi command mode and to exit the editor. These are described below:

Ex Escape - :

Syntax: :

Function: Enter Ex escape mode to execute an Ex command. The colon appears on the status line as a prompt for an Ex command. You then can enter an Ex command line terminated by either a <RETURN> or an <ESC> and the Ex command will execute. You then will be prompted to press <RETURN> to return to Vi command mode. During the input of the Ex command line or during execution of the Ex command you may press <INTERRUPT> to abort what you are doing and return to Vi command mode.

Exit Editor - ZZ

Syntax: **ZZ**

Function: Exit Vi and write out the file if any changes have been made. This returns you to the shell from which you invoked Vi.

Quit to Ex - Q

Syntax: Q

Function: Enter the Ex editor. When you do this, you will still be editing the same file. You can return to Vi by typing the "vi" command from Ex.

6.6 Ex Commands

Typing the colon (:) escape command when in command mode, produces a colon prompt on the status line. This prompt is for a command available in the line-oriented editor, Ex. In general, Ex commands let you write out or read in files, escape to the shell, or switch editing files.

Many of these commands perform actions that affect the "current" file by default. The current file is normally the file that you named when you invoked Vi, although the current file can be changed with the "file" command, **f**, or with the "next" command, **n**. In most respects, these commands are identical to similar commands for the editor, **ed**. All such Ex commands are terminated by either a <RETURN> or an <ESC>. We shall use a <RETURN> in our examples. Command entry is terminated by typing an <INTERRUPT>.

6.6.1 Command Structure

Most Ex command names are English words, and initial prefixes of the words are acceptable abbreviations. In descriptions, only the abbreviation is discussed, since this is the most frequently used form of the command. The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the command **substitute** can be abbreviated s while the shortest available abbreviation for the **set** command is **se**.

Most commands accept prefix addresses specifying the lines in the file that they are to effect. A number of commands also may take a trailing count specifying the number of lines to be involved in the command. Counts are rounded down if necessary. Thus, the command "10p" will print the tenth line in the buffer while "move 5" will move the current line after line 5.

Some commands take other information or parameters, stated after the command name. Examples might be option names in a **set** command, such as "set number", a filename in an **edit** command, a regular expression in a **substitute** command, or a target address for a **copy** command, such as

```
1,5 copy 25
```

Also, a number of commands have two distinct variants. The variant form of the command is invoked by placing an exclamation mark (!) immediately after the command name. Some of the default variants may be controlled by options;

in this case, the exclamation mark turns off the meaning of the default.

In addition, many commands take flags, including the characters "#", "p" and "l". A "p" or "l" must be preceded by a blank or tab. In this case, the command abbreviated by these characters is executed after the command completes. Since Ex normally prints the new current line after each change, p is rarely necessary. Any number of plus (+) or minus (-) characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

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

6.6.2 Command Addressing

The following specifies the line addressing syntax for Ex commands:

.	The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus "." is rarely used alone as an address.
n	The nth line in the editor's buffer, lines being numbered sequentially from 1.
$	The last line in the buffer.
%	An abbreviation for "1,$", the entire buffer.
+n or -n	An offset, n relative to the current buffer line. The forms ".+3" "+3" and "+++" are all equivalent. If the current line is line 100 they all address line 103.
/pat/ or ?pat?	Scan forward and backward respectively for a line containing pat, a regular

expression. The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing pat, then the trailing slash (/) or question mark (?) may be omitted. If pat is omitted or explicitly empty, then the string matching the last specified regular expression is located. The forms "/<RETURN>" and "?<RETURN>" scan using the last named regular expression. After a substitute, "//<RETURN>" and "??<RETURN>" would scan using that substitute's regular expression.

'' or 'x

Before each non-relative motion of the current line dot (.), the previous current line is marked with a label, subsequently referred to with two single quotes (''). This makes it easy to refer or return to this previous context. Marks are established with the Vi m command, using a single lowercase letter as the name of the mark. Marked lines are later referred to with the notation

'x.

where x is the name of a mark.

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

6.6.3 Command Format

The following is the format for all Ex commands:

[address] [command] [!] [parameters] [count] [flags]

All parts are optional depending on the particular command and its options. Command descriptions follow.

6.6.4 Argument List Commands

The argument list commands allow you to easily work on a set of files. This is done by remembering the list of filenames that are specified when you invoke Vi. The **args** command lets you examine this list of filenames. The **file** command gives you information about the current file. The **n** (next) command lets you edit the next file in the argument list or change the list. And the **rewind** command lets you restart editing the files in the list. All of these commands are described below:

args The members of the argument list are printed, with the current argument delimited by brackets. For example, a list might look like this:

file1 file2 [file3] file4 file5

Here, the current file is file3.

f Prints the current filename, whether it has been modified since the last **write** command, whether it is readonly, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line. In the rare case that the current file is "[Not edited]" this is noted also; in this case you have to use the form "w!" to write to the file, since the editor is not sure that a **w** command will not destroy a file unrelated to the current contents of the buffer.

f file The current filename is changed to file which is considered "[Not edited]".

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

n! The variant suppresses warnings about the modifications to the buffer not having been written out, discarding irretrievably any changes

which may have been made.

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

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

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

6.6.5 Edit Commands

To edit a file other than the one you are currently editing, you will often use one of the variations of the e command.

In the following discussions, note that the name of the current file is always remembered by Vi and is specified by a percent sign (%). The name of the previous file in the editing buffer is specified by a number sign (#). Thus, to edit the last file in the editing buffer, you could type:

```
:e #
```

The edit commands are described below:

e file
: Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last **w** command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After ensuring that this file is sensible, (i.e., that it is not a binary file, directory, or a device), the editor reads the file into its buffer. If the read of the file completes without error, the number of lines and characters read is printed on the status line. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered edited. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued.

The current line is initially the first line of the file.

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

e +n file
Causes the editor to begin editing at line n rather than at the first line. The argument n may also be an editor command containing no spaces; for example, "+/pat".

<CONTROL-^> This is a short-hand equivalent for ":e #<RETURN>", which returns to the previous position in the last edited file. If you do not want to write the file you should use ":e! #<RETURN>" instead.

6.6.6 Write Commands

The write commands let you write out all or part of your editing buffer to either the current file or to some other file. These commands are described below:

w file Writes changes made back to file, printing the number of lines and characters written. Normally, file is omitted and the text goes back where it came from. If a file is specified, then text will be written to that file. The editor writes to a file only if it is the current file and is edited, or if the file does not exist. Otherwise, you must give the variant form **w!** to force the write. If the file does not exist it is created. The current filename is changed only if there is no current filename; the current line is never changed.

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

w>> file Appends the buffer contents at the end of an existing file. Previous file contents are not destroyed.

w! name Overrides the checking of the normal **write** command, and writes to any file which the system permits.

w !command
Writes the specified lines into command. Note the difference between **w!** which overrides checks and **w !** which writes to a command. The output of this command is displayed on the screen and not inserted in the editing buffer.

6.6.7 Read Commands

The read commands let you read text into your editing buffer at any location you specify. The text you read in must be at least one line long, and can be either a file or the output from a command.

r file Places a copy of the text of the given file in the editing buffer after the specified line. If no file is given then the current filename is used. The current filename is not changed unless there is none, in which case the file becomes the current name. If the file buffer is empty and there is no current name then this is treated as an **e** command.

Address 0 is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the **e** command when the **r** successfully terminates. After an **r** the current line is the last line read.

r !command
Reads the output of command into the buffer after the specified line. A blank or tab before the exclamation mark (!) is mandatory.

6.6.8 Quit Commands

There are several ways to exit Vi. Some abort the editing session, some write out the editing buffer before exiting, and some warn you if you decide to exit without writing out the buffer. All of these ways of exiting are described below:

q Causes Vi to terminate. No automatic write of the editor buffer to a file is performed. However, Vi issues a warning message if the file has changed

since the last w command was issued, and does not quit. Vi will also issue a diagnostic if there are more files in the argument list left to edit. Normally, you will wish to save your changes, and you should give a **w** command. If you wish to discard them, use the "q!" command variant.

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

wq name Like a w and then a q command.

wq! name This variant overrides checking on the sensibility of the w command, as does "w!"

x name If any changes have been made and not written, writes the buffer out and then quits. Otherwise, it just quits.

6.6.9 Global and Substitute Commands

The global and substitute commands allow you to perform complex changes to a file in a single command. Learning how to use these commands is a must for the serious user of Vi. See also Section 6.8, "Regular Expressions."

g/pat**/**cmds

The g command has two distinct phases. In the first phase, each line matching pat in the editing buffer is marked. Next, the given command list is executed with dot (.) initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a backslash (\). This multiple-line option will not work from within Vi, you must switch to Ex to do it. If cmds (or the trailing slash (/) delimiter) is omitted, then each line matching pat is printed.

The g command itself may not appear in cmds. The options autoprint and autoindent are inhibited during a global command and the value of the report option is temporarily infinite, in deference to a report for the entire global. Finally, the context mark (' or `) is set to the value of dot (.) before the global command begins and is not changed during a global command.

The following list of global commands, most of them substitutions, covers the most frequent uses of the global command. These examples are well worth studying.

g/s1/p
: This command simply prints all lines that contain the string "s1".

g/s1/s//s2/
: This command substitutes the first occurrence of "s1" on all lines that contain it with the string "s2".

g/s1/s//s2/g
: This command substitutes all occurrences of "s1" with the string "s2". This includes multiple occurrences of "s1" on a line.

g/s1/s//s2/gp
: This command works the same as the preceding example, except that in addition, all changed lines are printed on the screen.

g/s1/s//s2/gc
: This command asks you to confirm that you want to make each substitution of the string "s1" with the string "s2". If you type a "y" then the substitution is made, otherwise it is not.

g/s0/s/s1/s2/g
: This command marks all those lines that contain the string "s0", and then for those lines only, it substitutes all occurrences of the string "s1" with "s2".

g!/pat/cmds
: This variant form of g runs cmds at each line not matching pat.

s/pat/repl/options
: On each specified line, the first instance of pattern pat is replaced by replacement pattern repl. If the global indicator option character "g" appears, then all instances are substituted. If the confirm indication character "c" appears, then before each substitution the line to be substituted is printed on the screen with the

string to be substituted marked with circumflex (^) characters. By typing a "y", you cause the substitution to be performed; any other input causes no change to take place. After an s command the current line is the last line substituted.

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

6.6.10 Text Movement Commands

The text movement commands are largely superseded by commands available in Vi command mode. However, the following two commands are still quite useful.

co addr flags
A copy of the specified lines is placed after addr, which may be "0". The current line "." addresses the last line of the copy.

[range]maddr
The **m** command moves the specified lines specified by range to be after addr. For example, "m+" swaps the current line and the following line, since the default range is just the current line. The first of the moved lines becomes the current line (dot).

6.6.11 Shell Escape Commands

You will often want to escape from the editor to execute normal XENIX commands. You may also want to change your working directory so that your editing can be done with respect to a different working directory. These operations are described below:

cd directory
The specified directory becomes the current directory. If no directory is specified, the current value of the home option is used as the target directory. After a **cd** the current file is not considered to have been edited so that write restrictions on pre-existing files still apply.

sh
A new shell is created. You may invoke as many commands as you like in this shell. To return to

Vi, you must type a <CONTROL-D> to terminate the shell.

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

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic is printed before the command is executed as a warning. A single exclamation (!) is printed when the command completes.

6.6.12 Other Commands

The following command descriptions explain how to use miscellaneous Ex commands that do not fit into the above categories:

nu

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed. To get automatic line numbering of lines in the buffer, set the number option.

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a w command has resulted in an error and you don't know how to save your work.

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

recover file

Recovers file from the system save area. The system saves a copy of the editing buffer only if you have made changes to the file, the system crashes, or you execute a preserve command. Except when you use preserve you will be notified by mail when a file is saved.

set argument

With no arguments, set prints those options whose values have been changed from their defaults; With the argument all it prints all of the option values.

Giving an option name followed by a question mark (?) causes the current value of that option to be printed. The "?" is unnecessary unless the option is Boolean valued. Switch options are given values either by the form "set option" to turn them on or "set ooption" to turn them off; string and numeric options are assigned via the form "set option=value".

More than one parameter may be given to set; they are interpreted left to right. for more information, see Section 6.7, "Start-Up Files and Options."

tag label

The focus of editing switches to the location of label. If Vi has to, it will switch to a different file in the current directory to find label. If you have modified the current file before giving a tag command, you must first write it out. If you give another tag command with no argument, then the previous label is used.

Similarly, if you type only a <CONTROL-]>, Vi searches for the word immediately after the cursor as a tag. This is equivalent to typing ":ta", this word, and then a <RETURN>.

The tags file is normally created by a program such as ctags, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag. This field is usually a contextual scan using

/pat/

to be immune to minor changes in the file. Such scans are always performed as if the nomagic option was set. The tag names in the tags file must be sorted alphabetically.

6.7 Start-Up Files and Options

There are a number of options that can be set to affect the Vi environment. These can be set with the Ex **set** command either while editing or immediately after Vi is invoked in the Vi start-up file, .exrc.

The first thing that must be done, however, before you can use Vi, is to set the terminal type so that Vi understands how to talk to the particular terminal you are using. This is the subject of the next subsection.

6.7.1 Setting the Terminal Type

To run Vi, the shell variable TERM must be defined and exported to your shell environment. How you do this depends on which shell you are using. You can usually determine which shell you are using by examining the prompt character. The normal shell prompts with a dollar sign ($); the C shell prompts with a percent sign (%).

Once you have determined which shell you are using, you need to find the name of your terminal type. For these examples, we will suppose that you are using an HP 2621 terminal. In the file /etc/termcap is a description of the capabilities of this terminal. Each terminal capability description has a unique name that corresponds to the type of a terminal supported by Vi. For the HP 2621, this "termcap" name is "2621".

6.7.1.1 The Normal Shell

To set your terminal type to 2621 you would place the following commands in the file .profile:

```
TERM=2621
export TERM
```

There are various ways of having this dynamically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a another hardwired terminal. One way is to place the following sequence of commands

```
tset -s -d 2621 > tset$$
. tset$$
rm tset$$
```

in your .profile file. The above line says that if you are

dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list. For the above sequence of commands to work, be sure that in your .profile file you have first set your PATH variable so that it includes the current directory

6.7.1.2 The C Shell

To set your terminal type to 2621 for the C shell, you would place the following commands in the file .login:

```
setenv TERM 2621
```

To specify your terminal type dynamically when you log in, you could use a procedure parallel to that discussed above for the normal shell:

```
tset -s -d 2621 > tset$$
source tset$$
rm tset$$
```

Place these commands in the file .login.

6.7.2 The .exrc File

Each time Vi is invoked, it reads commands from the file named .exrc in your home directory. This file normally sets the user's preferred options so that this need not be done each time Vi is invoked. A sample .exrc file follows:

```
set number
set wrapmargin=20
set errorbells
set ignorecase
set autoindent
```

Each of the above options is described in more detail below.

6.7.3 Options

There are only two kinds of options: switch options and string options. A switch option is either on or off. A switch is turned off by prefixing the word no to the name of the switch within a **set** command. String options are strings of characters that are assigned values with the syntax option=string. Multiple options may be specified on a line. Vi options are listed below:

autoindent, ai default: noai
Can be used to ease the preparation of structured program text. For each line created by an append, change, insert, open, or substitute operation, Vi looks at the preceding line to determine and insert an appropriate amount of indentation. To back the cursor up to the preceding tab stop one can press <CONTROL-D>. The tab stops going backwards are defined at multiples of the shiftwidth option. You cannot backspace over the indent, except by typing a <CONTROL-D>.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the autoindent is discarded.) Also specially processed in this mode are lines beginning with an up-arrow (^) and immediately followed by a <CONTROL-D>. This causes the input to be repositioned at the beginning of the line, but retains the previous indent for the next line. Similarly, a "0" followed by a <CONTROL-D> repositions the cursor at the beginning but without retaining the previous indent. Autoindent doesn't happen in global commands.

autoprint ap default: ap
Causes the current line to be printed after each Ex copy, move, or substitute command. This has the same effect as supplying a trailing "p" to each such command. Autoprint is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw
Causes the contents of the buffer to be written to the current file if you have modified it and give an next, rewind, tag, or ! command, or a <CONTROL-^> (switch files) or <CONTROL-]> (tag go to) command.

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

directory, dir default: dir=/tmp
Specifies the directory in which Vi places the editing buffer file. If this directory is not

writable, then the editor will exit abruptly when it fails to write to the buffer file.

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 be as in the command, instead of like &.

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

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

lisp default: nolisp
Autoindent indents appropriately for LISP code, and the ( ) { } [[and]] commands are modified to have meaning for LISP.

list default: nolist
All printed lines will be displayed unambiguously, showing tabs and end-of-lines.

magic default: magic
If nomagic is set, the number of regular expression metacharacters is greatly reduced, with only up-arrow (^) and dollar sign ($) having special effects. In addition the metacharacters "~" and "&" in replacement patterns are treated as normal characters. All the normal metacharacters may be made magic when nomagic is set by preceding them with a backslash (\).

mesg default: nomesg
Causes write permission to be turned off to the terminal while you are in visual mode, if nomesg is set. This prevents people writing to your screen with the XENIX write command and scrambling your screen as you edit.

number, n default: nonumber
Causes all output lines to be printed with their line numbers.

optimize, opt default: optimize
Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one line of output, thus greatly speeding output on terminals without addressable cursors when text with leading whitespace is printed.

paragraphs, para default: para=IPLPPPQPP LIbp
Specifies the paragraphs for the { and } operations. The pairs of characters in the option's value are the names of the nroff macros which start paragraphs.

redraw default: noredraw
The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal. Useful only at very high speed.

report default: report=5
Specifies a threshold for feedback from commands. Any command that modifies more than the specified number of lines will provide feedback as to the scope of its changes. For global commands and the undo command which have potentially 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 g command on the individual commands performed.

scroll default: scroll=1/2 window
Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode z command (double the value of scroll).

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

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

shiftwidth, sw default: sw=8
Gives the width of a software tab stop, used in reverse tabbing with <CONTROL-D> when using autoindent to append text, and by the shift commands.

showmatch, sm default: nosm
When a) or } is typed, move the cursor to the matching (or { for one second if this matching character is on the screen. Useful with LISP.

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

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

tags default: tags=tags /usr/lib/tags
A path of files to be used as tag files for the tag command. A requested tag is searched for in the specified files, sequentially. By default files named tag are searched for in the current directory and in /usr/lib.

term default=value of shell TERM variable
The terminal type of the output device.

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

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

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

w300, w1200, w9600
These are not true options but set window (above) only if the speed is slow (300), medium (1200), or high (9600), respectively.

wrapscan, ws default: ws
Searches using the regular expressions in

addressing will wrap around past the end of the file.

wrapmargin, wm default: wm=0
Defines a margin for automatic wrap over of text during input.

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

6.8 Regular Expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. Vi remembers two previous regular expressions: the previous regular expression used in a substitute command and the previous regular expression used elsewhere, referred to as the previous scanning regular expression. The previous regular expression can always be referred to by a null regular expression: e.g., "//" or "??".

The regular expressions allowed by Vi are constructed in one of two ways depending on the setting of the magic option. The Ex and Vi default setting of magic gives quick access to a powerful set of regular expression metacharacters. The disadvantage of magic is that the user must remember that these metacharacters are magic and precede them with the character "\" to use them as "ordinary" characters. With nomagic set, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the now ordinary character with a "\". Note that "\" is thus always a metacharacter. In this discussion the magic option is assumed. With nomagic the only special characters are "^" at the beginning of a regular expression, dollar sign ($) at the end of a regular expression, and backslash (\). The tilde (~) and the ampersand (&) also lose their special meanings related to the replacement pattern of a substitute.

The following basic constructs are used to construct magic mode regular expressions.

char
An ordinary character matches itself. Ordinary characters are any characters except a circumflex (^) at the beginning of a line, a dollar sign ($) at the end of line, an asterisk (*) as any character other than the

first, and any of the following characters:

. \ [~

These characters must be escaped (i.e., preceded) by a backslash (\) if they are to be treated as ordinary characters.

^ At the beginning of a pattern this forces the match to succeed only at the beginning of a line.

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

. Matches any single character except the newline character.

\< Forces the match to occur only at the beginning of a "word;" that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> Similar to "\<", but matching the end of a "word," i.e. either the end of the line or before a character which is not a letter, a digit, or the underline character.

[string] Matches any single character in the class defined by string. Most characters in string define themselves. A pair of characters separated by a dash (-) in string defines the set of characters between the specified lower and upper bounds, thus "[a-z]" as a regular expression matches any single lowercase letter. If the first character of string is a circumflex (^) then the construct matches those characters which it otherwise would not. Thus "[^a-z]" matches anything but a lowercase letter or a newline. To place any of the characters caret, left bracket, or dash in string they must be escapes with a preceding backslash (\).

The concatenation of two regular expressions first matches the leftmost regular expression and then the longest string which can be recognized as a regular expression. The first part of this new regular expression matches the first regular expression and the second part matches the second.

Any of the single character matching regular expressions mentioned above may be followed by an asterisk or "star" (*) to form a regular expression which matches zero or more adjacent occurrences of the characters matched by the prefixing regular expression. The tilde (~) may be used in a regular expression, and matches the text which defined the replacement part of the last s command. A regular expression may be enclosed between the sequences "\(" and "\)" with side effects in substitute replacement patterns.

The basic metacharacters for the replacement pattern are the ampersand (&) and the asterisk (*) these are given as "\&" and "\~" when nomagic is set. Each instance of the ampersand is replaced by the characters matched by the regular expression. The metacharacter "~" stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by a backslash (\). The sequence "\n" is replaced by the text matched by the n-th regular subexpression enclosed between "\(" and "\)". When nested, parenthesized subexpressions are present, n is determined by counting occurrences of "\(" starting from the left. The sequences "\u" and "\l" cause the immediately following character in the replacement to be converted to upper or lowercase, respectively, if this character is a letter. The sequences "\U" and "\L" turn such conversion on, either until "\E" or "\e" is encountered, or until the end of the replacement pattern.

6.9 Speeding Things Up

At times, getting in and out of Vi can take much more time than you'd like. To keep the speed of invocation down, and to keep the number of invocations to a minimum, the following hints are offered:

- Keep files around 20,000 characters in size, or smaller. Files larger than 100,000 characters are intractable.

- Invoke Vi with a list of filename arguments. You can then use the next (:n) command to cycle through the named files. This is faster and more efficient than using a series of edit (:e) commands or invoking the editor for each file,

- When transferring text between files, yank or delete text into a buffer; switch files, and then "put" the

text into the new file.

- If you want to execute XENIX commands while in Vi, use the shell escape ":!", or the ":sh" command to create a new shell. With either method, you can quickly return to Vi without reinvocation.

6.9.1 When To Use Ex

Since Ex and Vi are one and the same program, you can switch back and forth between the two quite easily. To enter Ex from Vi, use the Q command. To enter Vi from Ex, use the **vi** command.

One major difference between the two is that Ex is line-oriented and uses XENIX standard input and output. Vi is screen-oriented and uses its own special input routines for handling keyboard input.

Because of this difference, Ex is in some cases better than Vi. For example, you can write scripts only for Ex: scripts won't work in Vi. Similarly, if you know how to use the line editor, Ed, the transition to Ex will be easy, since the Ex command set is largely a superset of Ed. Also use Ex whenever you need to perform a substitution command that requires more than one line of input. Newlines can be escaped with a backslash (\) in Ex, but not in Vi. Therefore, the following command is legal in Ex, but not in Vi:

```
s/one line/one line\
two lines\
three lines\
four lines/
```

6.10 Limitations

In using Vi, you should note the following limits:

- 250,000 lines in a file
- 1024 characters per line
- 256 characters per global command list
- 128 characters per filename
- 128 characters in the previous inserted and deleted text

- 100 characters in a shell escape command
- 63 characters in a string valued option
- 30 characters in a tag name

6.11 Troubleshooting

The following is a list of common problems that you may encounter when using Vi, along with the probable solution for each.

Don't know which operation mode you're in
: You often want to abort a command or exit insert mode. To do this, type <ESC> until the bell rings to assure your return to Vi command mode.

Can't get out of sub-shell
: Type <CONTROL-D> to exit any shell.

Accidentally entered Ex command mode
: If you pressed colon (:) accidentally, you can return to where you were in your file by typing an <ESC>.

Inadvertent deletion or insertion
: Type "u" to undo the last delete or insert command.

Scrambled Screen
: Type <CONTROL-L> to redraw the screen.

Line too long
: Go to the middle of the line and type

```
r<RETURN>
```

: to break the line into two separate lines.

Can't see control characters
: Set the list option by typing:

```
:set list
```

: This will substitute a circumflex (^) and the name of the control letter in place of each control character. This is most useful for detecting tab characters in files. For example, "^I" would appear instead of the expansion of the tab into one or more spaces.

Not in screen mode

Type ":q!" to abort and then restart. Check to see if the TERM variable in the file .profile in your home directory contains the proper terminal type definition.

Mistakenly in open mode

See above.

Full file not read in

Type ":e! %" to re-edit the current file.

Keyboard locked up

Vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, type:

```
stty -nl echo -cbreak <LF>
```

Typing a linefeed, i.e., a <LF> or <CONTROL-J>) is important here, since it is quite possible that the <RETURN> key will not work as a newline character.

System crashed

Normally, Vi will inform you that your editing buffer has been saved before a crash. The buffer can be recovered by typing

```
vi -r filename
```

If Vi was unable to save the buffer before the crash, then the editing buffer is irretrievably lost.

6.12 Character Functions

This section describes the use of each key on your keyboard in Vi command mode and in insert mode. It does not describe the use of these keys in Ex command mode. If a character's only meaning in insert mode is to insert the character in the text as-is, then only its meaning as a command is discussed. Characters are presented alphabetically with a lowercase, an uppercase, and a control character description for each letter of the alphabet.

a Appends text after the current cursor position.

A Appends text at the end of the current line. This is a synonym for "$a".

<CONTROL-A> Not used as a command.

<BKSP> Moves the cursor one character to the left. This has the same function as the left arrow key, the "h" key, and <CONTROL-H>. Note that arrow keys on certain kinds of terminals cannot be used, and that some terminals have no arrow keys at all. A preceding count repeats the effect.

In insert mode, <BKSP> eliminates the last input character, backing over it and erasing it from the insertion, but not from the screen. The character remains so that you can see what you have typed if you wish to type something slightly different.

b Backs up the cursor to the beginning of the current word. A word is a sequence of alphanumeric characters, or a sequence of special characters. A preceding count repeats the effect.

B Backs up a word and places the cursor at the beginning of the current word, where words are composed of non-blank sequences, A preceding count repeats the effect.

<CONTROL-B> Move the cursor back one screen window. A preceding count repeats the effect. Two lines from the preceding window are kept for continuity, if possible.

c Changes the following text object, replacing it with input text up to an <ESC>. If more than part of a single line is affected, the text which is deleted is saved in the numeric named buffers. If only part of the current line is affected, then the last character in the text to be changed is marked with a dollar sign ($). A count specifies the number of the given objects to be affected, thus both "3c)" and "c3)" change three sentences.

C Changes the rest of the text on the current line, replacing it with input text up to an <ESC>. This is a synonym for "c$".

<CONTROL-C> Not used as a command.

d Deletes the following text object. If more than part of a line is affected, the text is saved in the numeric buffers. A count specifies the number of the given objects to be affected, thus "3dw" is the same as "d3w".

D Deletes text from the cursor to the end of the line. The character beneath the cursor is deleted as part of this command. This is a synonym for "d$".

<CONTROL-D> Scrolls down a half-window of text. A preceding count gives the number of (logical) lines to scroll, and is remembered for future <CONTROL-D> and <CONTROL-U> commands. When in insert mode, <CONTROL-D> backtabs over autoindent white space at the beginning of a line: this white space cannot be backspaced over.

e Advances to the end of the next word, where a word is defined as for the **b** and **w** commands. A preceding count repeats the effect.

E Moves forward to the end of a word, where a word is defined as for the **B** and **W** commands. A preceding count repeats the effect.

<CONTROL-E> Not used as a command.

<ESC> Exits insert mode and cancels a partially formed command, such as a **z** when no following character has yet been given. Also

terminates command line input on the status line for commands such as colon (:), slash (/), and question mark (?)). If an <ESC> is given when in command mode, the editor rings the bell or flashes the screen.

f Finds the first instance of a character following the cursor. The next typed character is the character given as an argument. The search for the character begins following the cursor on the current line. The search goes no farther than the end of line. A preceding count repeats the effect.

F Finds a single following character, preceding the cursor. The next typed character is the character given as an argument. The search for the character begins preceding the cursor on the current line. The search goes no farther than the beginning of the line. A preceding count repeats the effect.

<CONTROL-F> Move the cursor forward one screen window. A preceding count repeats the effect. Two lines from the preceding window are kept for continuity, if possible.

g Not used as a command.

G Moves the cursor to the line specified by a preceding line number argument (e.g., "12G" moves the cursor to the beginning of line 12. If no line number is given, then the cursor moves to the end of the file. The screen is redrawn with the new current line in the center of the screen.

<CONTROL-G> Equivalent to ":f<RETURN>", Prints the name of the file currently being edited, whether the file has been modified during the current editing session, the line number of the current line, the number of lines in the file, and the percentage of the file viewed in lines, relative to the current line.

h See <BKSP>.

H Homes the cursor to the beginning of the top line on the screen. If a count is given, then the cursor is moved to the beginning of

the line that is the "count'th" line from the top of the screen. In any case, the cursor is moved to the first non-whitespace character on the line.

<CONTROL-H> See **h**.

i Inserts text before the cursor, otherwise like **a**.

I Inserts text at the beginning of the current line. This is a synonym for "^i".

<CONTROL-I> See <TAB>.

j Moves the cursor one line down in the same column. If the column position does not exist, vi comes as close as possible to the same column. Synonyms include <CONTROL-J>, <LF>, and <CONTROL-N>.

J Joins together the current line with the next line, supplying appropriate white space: one space between words, two spaces after a period, and no spaces at all if the first character of the joined line is a right parenthesis. A count causes the given number of lines to be joined.

<CONTROL-J> See **j**.

k Moves the cursor up one line in the same column. <CONTROL-P> is a synonym.

K Not used as a command.

<CONTROL-K> Not used as a command.

l Moves the cursor one character to the right. The <SPACE> and right arrow keys are synonyms.

L Moves the cursor to the first non-white character of the last line on the screen. With a preceding count, it moves to the first non-white of the "count'th" line from the bottom.

<CONTROL-L> Causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program scramble the

screen, or after output is stopped by an interrupt. The <FF> or form feed key is a synonym for this key.

m Marks the current position of the cursor with an alphabetic "mark". The mark is specified by the character given as an argument to the command. This character is the next key typed, and must be in the range "a"-"z". Move to this marked position using ' or `.

M Moves the cursor to the middle line on the screen, at the first non-white position on the line.

<CONTROL-M> Same as <RETURN>.

n Repeats the last search command (/ or ?).

N Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of **n**.

<CONTROL-N> See **j**.

o Opens a new line below the current line so that input text can be inserted up to an <ESC>.

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.

<CONTROL-O> Not used as a command.

p Puts last deleted text at either the line below the cursor or inserts that text before the cursor on the same line. See P below for details.

P Puts the last deleted text back above or before 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. P may be preceded by a named buffer specification of the form

"x

to retrieve the contents of the named buffer. Buffers 1-9 contain deleted material; buffers "a"-"z" are available for general use.

<CONTROL-P> See **k**.

q Not used as a command.

Q Switches from Vi to Ex command mode. In Ex command mode, whole lines form commands, ending with a <RETURN>. You can give most Ex commands in Ex command mode without typing a preceding colon; the editor will supply the colon as a prompt. To return to Vi command mode, use the "vi" command.

<CONTROL-Q> Not a command character.

<RETURN> Advances to the next line, at the first non-white position in the line. If given a count, it advances that many lines.

When in insert mode, a <RETURN> causes the insertion to continue on to a new line. The <RETURN> key is the same as the <CONTROL-M> key.

r Replaces the single character at the cursor with the next typed character. The new character may be a <RETURN>: this is the easiest way to split lines. A count replaces each of the following count characters with the single character given.

R Replaces characters on the screen with characters you type (this is sometimes called overstrike mode). Replacement terminates with an <ESC>.

<CONTROL-R> Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines containing only a single at-sign (@)).

s Substitutes the single character under the cursor to the following inserted text. Text insertion is terminated with an <ESC>. If given a count, s then substitutes the given number of characters rather than just one.

The last character to be changed is marked with a dollar sign ($), as with the c command.

S Changes whole lines, a synonym for the cc command. A count substitutes the given number of lines instead of just one. Substituted lines are saved in the numeric buffers and erased on the screen before the substitution begins.

<CONTROL-S> Not used as a command.

<SPACE> See **l** above.

t Advances the cursor forward, up to, but not on a given character. The given character is the next character typed. Most useful with operators such as **d** and c to delete the characters up to a character. Use dot (.) to delete more if this doesn't delete enough the first time.

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.

<CONTROL-T> Not a command character. During an insertion, with autoindent set and at the beginning of the line, inserts shiftwidth white space.

<TAB> Not a command character. When in insert mode, a <TAB> prints as a number of spaces appropriate to vi's tab settings. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tab stops is controlled by the tabstop option.

u Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus this command is its own inverse. When used after an insertion that inserted text on more than one line, then the inserted lines are saved in the numeric named buffers.

U Restores the current line to its state before you started changing it. This works only for the duration of the particular line edit; not for the duration of the editing session.

<CONTROL-U> Scrolls the screen up. Counts work as they do for <CONTROL-D>. The previous scroll amount is common to both.

v Not used as a command.

V Not used as a command.

<CONTROL-V> Not a command character. In input mode, <CONTROL-V> quotes the next character so that it is possible to insert nonprinting and special characters into the file.

w Advances the cursor to the beginning of the next word, where a word is as defined in the b command, above.

W Moves the cursor forward to the beginning of a word in the current line, where words are defined as sequences of non-blank characters. A count repeats the effect.

<CONTROL-W> Not a command character. During an insertion, <CONTROL-W> backs up as b would in command mode.

x Deletes the single character under the cursor. With a count, **x** deletes the given number of characters forward from the cursor position, but only on the current line.

X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

<CONTROL-X> Not used as a command.

y "Yanks" the following object into the unnamed temporary buffer. If preceded by a named buffer specification, then the text is placed in that buffer also. Text can be recovered with a later **p** or P command.

Y "Yanks" a copy of the current line into the unnamed buffer, to be put back by a later **p** or P command. This is a useful synonym for

"yy". A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.

<CONTROL-Y> Not used as a command.

z Adjusts and redraws the screen with the current line placed as specified by an alignment character. The alignment character may be either a <RETURN>, a period (.), or a minus (-). A <RETURN> specifies the top of the screen, a dot the center of the screen, and a minus the bottom of the screen. A count may be given after the **z**, but before the alignment character, to specify the new screen size for the redraw. A count before the **z** gives the number of the line to place in the center of the screen instead of the default current line.

ZZ Exits the editor. (Same as ":x<RETURN>".) If any changes have been made, the buffer is written out to the current file. Then the editor quits.

<CONTROL-Z> Not used as a command.

0 Moves the cursor to the first character on the current line. Also used in forming numbers, after an initial 1-9. Note that no command can take an initial zero as part of a count argument.

1-9 Used to form numeric arguments to commands.

(Retreats to the beginning of a sentence, or a LISP s-expression, if the lisp option is set. A sentence ends at a period (.), exclamation (!), or question mark (?) which is followed by either the end of a line or by two spaces. Any number of closing parentheses, brackets, quotation marks, or single quotes may appear after the period, exclamation, or question mark, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[below). If given a count, the command is repeated. A count advances the given number of sentences.

) Advances to the beginning of a sentence. A count repeats the effect. See (above for

the definition of a sentence.

[[Backs up to the previous section boundary. A section begins at each macro in the sections option, normally a ".NH" or ".SH" and also at lines that start with a formfeed, <CONTROL-L>. Lines beginning with { also stop [[. This makes it useful for looking backwards, a function at a time, in C programs. If the lisp option is set, the [[stops at each left parenthesis [(] at the beginning of a line.

]] Move forward to a section boundary. See [[for the definition of a section boundary.

< An operator which shifts lines left one shiftwidth, normally eight spaces. Affects lines when repeated, as in "<<". Counts are passed through to the basic object, thus "3<<" shifts three lines.

> Shifts lines right one shiftwidth, normally eight spaces. Counts repeat the basic object. This operator can be made to affect a given number of lines when the character is doubled and preceded by the number of lines to be affected, as in "23>>".

{ Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the paragraphs option, normally at a ".IP", ".LP", ".PP", ".QP" and ".bp". A paragraph also begins after a completely empty line, and at each section boundary (see [[above).

} Advances to the beginning of the next paragraph. See { for the definition of paragraph.

` When followed by another back quote, this returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter in the range "a"-"z", returns to the position marked with the given letter by an **m** command. When used with an operator such as **d**, the operation takes place from the exact marked place to the current position within the line; If a single quote (') is used, the operation takes place over

whole lines instead.

' When followed by a single quote (') returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter in the range "a"-"z", it returns to the line marked with the given letter by an **m** command. When used with an operator such as **d**, the operation takes place over complete lines; if you use a back quote (`), the operation takes place from the exact marked place to the current cursor position within the line.

+ Same as <RETURN> when used as a command.

- Retreats to the previous line at the first non-white character. This is the inverse of plus (+) and <RETURN>. If the line moved to is not on the screen, the screen is scrolled, or, if this is not possible, cleared and redrawn. If a large amount of scrolling would be required the screen is cleared and redrawn, with the current line at the center.

^ Moves to the first non-white position on the current line.

_ Not used as a command.

! Processes a text object in the editing buffer with a XENIX command given as the second argument. The first argument is a cursor movement command that delimits the text object. The XENIX command is processed by the shell, so normal XENIX quoting conventions apply. The command line itself is terminated with a <RETURN>. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the **!**. Thus "2!}sort<RETURN>" sorts the next two paragraphs by running them through the program sort. 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 1-9 used for saving deleted text and named buffers "a"-"z" into which you

can place text.

$ Moves the cursor to the end of the current line. If you use ":set list<RETURN>", then the end of each line will be shown by printing a dollar sign ($) after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus "2$" advances to the end of the following line.

% Moves the cursor to the parenthesis or brace that balances the parenthesis or brace at the current cursor position.

& A synonym for ":&<RETURN>", by analogy with the Ex & command.

, Reverse the last f, F, t, or T command, searching in the opposite direction on the current line. This is especially useful after typing too many semicolon (;) characters. A count repeats the search.

; Repeats the last single character find command (either f, F, t, or T). A count repeats the scan count number of times.

. Repeats the last command that changed the buffer. This is especially useful when deleting words or lines; you can delete some words or lines and then press dot (.) to delete more words or lines. If given a count, "dot" passes it on to the command being repeated. Thus after a "2dw", "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 normal input editing sequences may be used during input on the status line.

The search begins when you press <RETURN>. The cursor then moves to the beginning of the last line to indicate that the search is in progress. The search can be terminated by typing an <INTERRUPT>. Searches normally wrap around the end of the file to the beginning to find a string anywhere in the buffer.

When used with an operator the enclosed region is normally affected. By giving 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 slash (/> and then an offset +n or -n.

? Searches backwards for the string given as an argument. This is the opposite of slash (/). See the description of slash above for details on searching.

: The Ex command escape character. Ex command input is entered on the status line. If you press colon (:) accidentally, you can return to where you were by typing an <INTERRUPT>.

| Places the cursor on the character in the column specified by a preceding count.

= Not used as a command.

@ Not used as a command.

~ Not used as a command.

* Not used as a command.

Not used as a command.

\ Not used as a command.

<CONTROL-_> Not used as a command. Reserved as the command character for some terminals.

<CONTROL-?> Interrupts the editor, returning it to command mode.

<CONTROL-@> Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insertion terminates. Only 128 characters are saved from the last insertion. If more characters were inserted than 128, then this command inserts no characters. A <CONTROL-@> cannot be part of a file, even if quoted.

<CONTROL-[> Same as <ESC>.

<CONTROL-]> Searches for the word which is after the cursor as a tag. Equivalent to typing ":ta",

this word, and then a <RETURN>. Mnemonically, this command is "go right to".

<CONTROL-^> Equivalent to ":e #<RETURN>", Returns to the previous position in the last edited file. If you do not want to write the file you should use ":e! #<RETURN>" instead.

CHAPTER 7

THE SHELL

CONTENTS

7.1 Introduction

When users log into XENIX, they communicate the shell command interpreter, sh. This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one single-minded function: to read and execute commands from its standard input.

Because the shell gives the user a high-level language in which to communicate with the operating system, XENIX can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With XENIX and the shell, commands can be:

- Combined to form new commands
- Passed positional parameters
- Added or renamed by the user
- Executed within loops or executed conditionally
- Created for local execution without fear of name conflict with other user commands
- Executed in the background without interrupting a session at a terminal

Furthermore, commands can "redirect" command input from one source to another and redirect command output to a file, terminal, printer, or to another command. What this provides the user is flexibility and generality; flexibility in tailoring a task for a particular purpose and generality in the concept of input and output.

7.2 XENIX and the Shell

The shell itself (i.e., the program that reads your commands when you log in or that is invoked with the sh command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

Note that the word "shell" has two distinct meanings when used in this chapter. Normally, when we refer to "the shell", we mean the interpreter that implements the shell command language; the word "shell" in this case is used in a general sense. However, when we talk about several instances of executing command interpreters, then each of these is called a "shell". Keep these two usages of the

word in mind as you read this chapter.

7.2.1 The Shell Command Interface

Because the shell is the interface to all of the resources of the operating system, it is important to understand the structure of this interface. Figure 7-1 illustrates the overall structure:

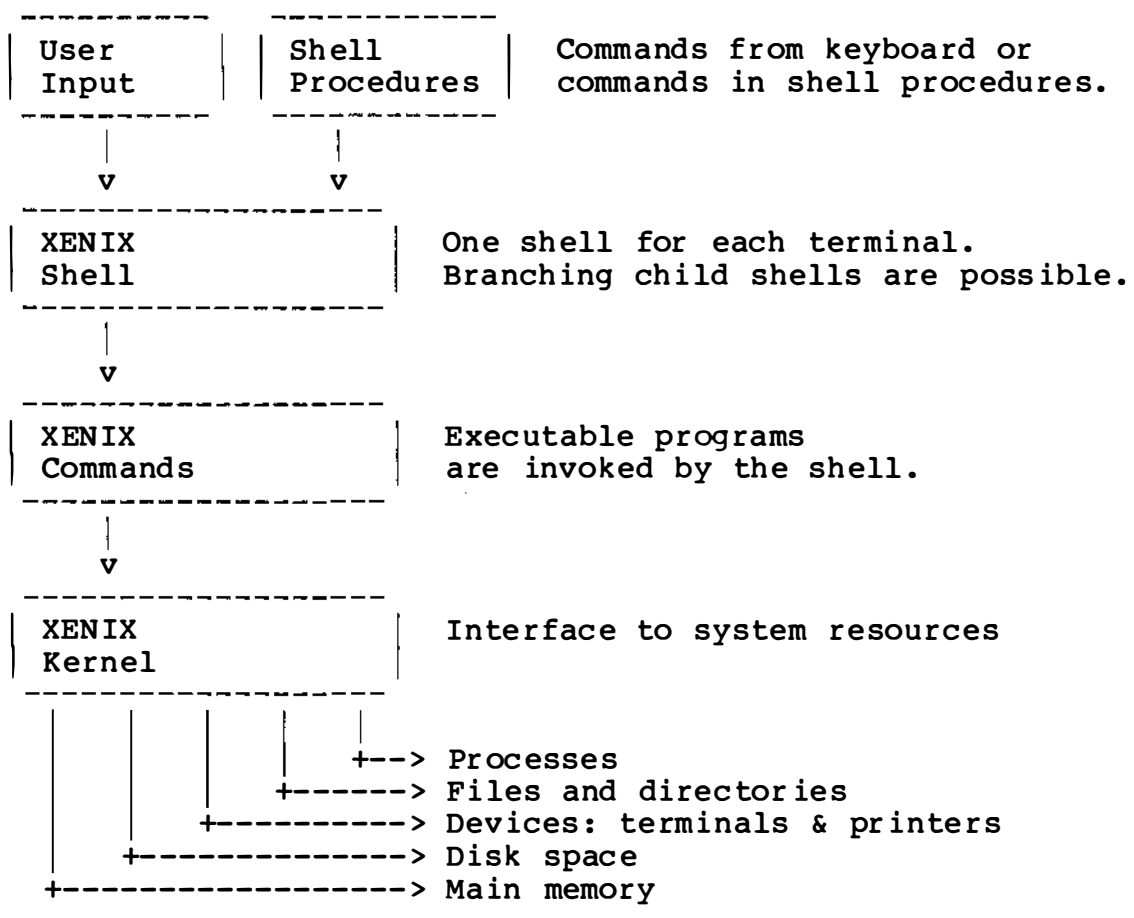


Figure 7-1. XENIX Command Structure

Commands are typed at the user's terminal or read from a file. Command lines are read and interpreted by the user's shell so that executable XENIX programs can be invoked. Invoked commands then make system calls to the XENIX kernel. The actions of the kernel are transparent to the user at the keyboard.

7.2.2 How Shells Are Created

In XENIX, a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and may even give birth to new life by "forking" new processes. Thus, at any given moment in XENIX, several processes may be executing, some of which are "children" of other processes. Users log into the operating system and are assigned a "shell" from which they execute. This shell is simply a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In this multi-tasking environment, processes may be created in one process and then sent off on their own to be processed in the "background." Such jobs are analogous to batch jobs in larger computer installations.

A schematic look at how user shells might fork other shells is shown in Figure 7-2.

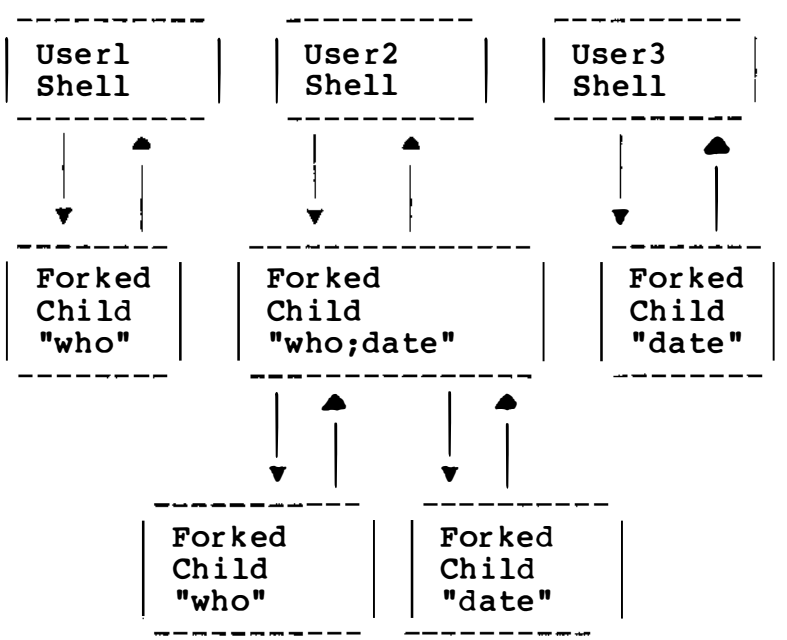


Figure 7-2. Forking Shells

Arrows show how control is passed from a parent shell to a child shell leaving the parent waiting. When a child shell completes execution, it dies and control returns to the parent shell.

7.3 Using the Shell

The most common way of using the shell is by typing simple commands at your keyboard. A simple command is any sequence of arguments separated by spaces or tabs. The first argument (numbered zero) specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. The following command line might be typed to request printing of the files allan, barry, and calvin:

```
lpr allan barry calvin
```

If the first argument of a command names a file that is executable (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell, as parent, creates a child process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a sub-shell) to read the file and execute the commands inside it. The shell forks to do this, but no exec call is made.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This provides uniformity of invocation and ease of choosing the implementation of a given command.

7.4 How the Shell Finds Commands

The shell normally searches for commands in three distinct locations in the file system. The shell first attempts to use the command name as given; if this fails, it prepends the string /bin to the name, and finally, /usr/bin. The effect is to search, in order, the current directory, then the directory /bin, and finally, /usr/bin. For example, the pr and **man** commands are actually the files /bin/pr and /usr/bin/man, respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If a given command name begins with a slash (/) (e.g., /bin/sort or /cmd), the prepending is not performed. Instead, a single attempt is made to execute the unmodified command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the shell PATH variable.

7.4.1 Generation of Argument Lists

The arguments to commands are very often filenames. Sometimes, these filenames have similar, but not identical names. To take advantage of this similarity in names, the shell lets the user specify patterns that match the filenames in a directory. If a pattern is matched by one or more filenames in a directory, then those filenames are automatically generated by the shell as arguments to the command.

Most characters in such a pattern match themselves, but there are also special metacharacters that may be included in a pattern. These special characters are: the asterisk (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters.

Inside brackets, a pair of characters separated by a dash (-) matches any character lexically within the range of that pair. Thus, "[a-de]" is equivalent to "[abcde]".

Examples follow of metacharacter usage:

```
*          (Matches all names in the current directory)
*temp*     (Matches all names containing "temp")
[a-f]*     (Matches all names beginning with "a" through "f")
*.c        (Matches all names ending in ".c")
/usr/bin/? (Matches all single-character names in /usr/bin)
```

This pattern matching capability saves much typing and, more importantly, makes it possible to organize information in large collections of small files that are named in a disciplined way.

Pattern-matching has some restrictions. If the first character of a filename is a period (.), it can be matched

only by an argument that literally begins with a period. If a pattern does not match any filenames, then the pattern itself is returned as the result of the match.

Note that directory names should not contain any of the following characters:

```
* ? [ ]
```

because this may cause infinite recursion during pattern matching attempts.

7.4.2 Quoting Mechanisms

Many characters have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. Single quotes (') and double quotes (") surrounding a string, or a backslash (\) before a single character, provide this function in somewhat different ways. (Back quotes (`) are used only for command substitution in the shell and do not hide the special meanings of any characters.)

Within single quotes, all characters (except the single quote (') itself) are taken literally, with any special meaning removed. Thus

```
echostuff='echo $? $*; ls * | wc'
```

results in the string

```
echo $? $*; ls * | wc
```

being assigned to the variable echostuff, but it does not result in any other commands being executed.

Within double quotes, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign ($), the single quote (`), and the double quote (") itself. Thus, within double quotes, variables are expanded and command substitution (discussed in a later section) takes place. However, any commands in a command substitution are not affected by double quotes outside of the grave accents, so that characters such as star (*) retain their special meaning.

To hide the special meaning of dollar sign ($), single quote (`), and double quote (") within double quotes, precede these characters with a backslash (\). Outside of double

quotes, preceding a character with a backslash is equivalent to placing single quotes around that character. A backslash (\) followed by a newline causes that newline to be ignored and is equivalent to a space. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

7.5 Redirection of Input and Output

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency's sake, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

7.5.1 Standard Input and Output

When a command begins execution, it usually expects that three files are already open: a "standard input," a "standard output," and a "diagnostic output," (also called "standard error"). A number called a file descriptor is associated with each of these files. By convention, file descriptor 0 is associated with the standard input, file descriptor 1 with the standard output, and file descriptor 2 with the diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the terminal screen). The shell permits them to be redirected elsewhere before control is passed to an invoked command.

An argument to the shell of the form "<file" or ">file" opens the specified file as the standard input or output (in the case of output, destroying the previous contents of file, if any). An argument of the form ">>file" directs the standard output to the end of file, thus providing a way to append data to it without destroying its existing contents. In either of the two output cases, the shell creates file if it does not already exist (thus ">output" alone on a line creates a zero-length file). The following appends to file log the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor

pattern matching of filenames occurs after these substitutions. For example

```
echo 'this is a test' > *.gal
```

produces a one-line file named *.gal, a most unfortunate name for a file. Similarly, an error message is produced by the following command, unless you have a file with the name "?":

```
cat < ?
```

So remember, special characters are not expanded in redirection arguments. The reason this is so, is that redirection arguments are scanned by the shell before pattern recognition and expansion takes place.

7.5.2 Diagnostic and Other Outputs

Diagnostic output from XENIX commands is traditionally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines.) One can redirect this error output to a file by immediately prepending the number of the file descriptor (i.e., 2 in this case) to either output redirection symbol (> or >>). The following line appends error messages from the **cc** command to the file named ERRORS:

```
cc testfile.c 2>>ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol without any intervening spaces or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if cmd puts output on file descriptor 9, then the following line will capture that output in the file savedata:

```
cmd 9>savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose, for example, that cmd directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file

descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard   2>error   9>data
```

7.5.3 Command Lines and Pipelines

A sequence of commands separated by the vertical bar (|) makes up a pipeline. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by pipes, i.e., the output of each command (except the last one) becomes the input of the next command in line.

A filter is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, they are synchronized to the extent that each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read larger amounts of data before producing output; sort is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline, where nroff is a text formatter whose output may contain reverse line motions, col converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and lpr does the actual printing. The flag -mm indicates one of the commonly used formatting options, and text is the name of the file to be formatted:

```
nroff -mm text | col | lpr
```

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these examples at a terminal:

- who
  Prints the list of logged-in users on the terminal screen.

- who >>log
  Appends the list of logged-in users to the end of file log.

- who | wc -l
  Prints the number of logged-in users. (The argument to **wc** is "minus ell".)

- who | pr
  Prints a paginated list of logged-in users.

- who | sort
  Prints an alphabetized list of logged-in users.

- who | grep bob
  Prints the list of logged-in users whose login names contain the string bob.

- who | grep bob | sort | pr
  Prints an alphabetized, paginated list of logged-in users whose login names contain the string bob.

- { date; who | wc -l } >>log
  Appends (to file log) the current date followed by the count of logged-in users. Be sure to place a space after the left brace.

- who | sed -e 's/ .*//' | sort | uniq -d
  Prints only the login names of all users who are logged in more than once. Note the use of **sed** as a filter to remove characters trailing the login name from each line.

The **who** command does not by itself provide options to yield all these results -- they are obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace "who | " with "</etc/passwd" in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line, even at the start.

7.5.4 Command Substitution

Any command line can be placed within back quotes (`...`) so that the output of the command replaces the command line itself. This concept is known as command substitution. The command or commands enclosed between back quotes are first executed by the shell and then their output replaces the whole expression, back quotes and all. This feature is often used to assign to shell variables. (Shell variables are described in the next section.) For example

```
today=`date`
```

assigns the string representing the current date to the variable today (e.g., "Tue Nov 27 16:01:09 EST 1982"). The following command saves the number of logged-in users in the variable users:

```
users=`who | wc -l`
```

Any command that writes to the standard output can be enclosed in back quotes. Back quotes may be nested, but the inside sets must be escaped with backslashes (\). For example:

```
logmsg=`echo Your login directory is \`pwd\``
```

Shell variables can also be given values indirectly by using the read command. The read command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example

```
read first init last
```

takes an input line of the form

```
G. A. Snyder
```

and has the same effect as typing:

```
first=G.   init=A.   last=Snyder
```

The read command assigns any excess "words" to the last variable.

7.6 Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as positional parameters; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

7.6.1 Positional Parameters

When a shell procedure is invoked, the shell implicitly creates positional parameters. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter $0. The first command argument is called $1, and so on. The **shift** command may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the **set** command. For example

```
set abc def ghi
```

assigns the string "abc" to the first positional parameter, $1, the string "def" to $2, and the string "ghi" to $3. Note that $0 may not be assigned a value in this way -- it always refers to the name of the shell procedure, or in the login shell, to the name of the shell.

7.6.2 User-Defined Variables

The shell also recognizes alphanumeric variables to which string values may be assigned. A simple assignment has the syntax:

```
name=string
```

Thereafter, $name will yield the value string. A name is a sequence of letters, digits, and underscores that begins with a letter or an underscore. No spaces surround the equals sign (=) in an assignment statement. Note that positional parameters may not appear on the left side of an assignment statement; they can only be set as described in the previous section.

More than one assignment may appear in an assignment statement, but beware: the shell performs the assignments from right to left.

The following command line results in the variable a acquiring the value "abc":

```
a=$b b=abc
```

The following are examples of simple assignments. Double quotes around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, while also allowing variable substitution (also known as

"parameter substitution") to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign ($) are replaced by the corresponding values, if any. Single quotes inhibit variable substitution:

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable echovar has as its value the string consisting of the values of the first four positional parameters, separated by spaces. No quotes are needed around the string of asterisks being assigned to stars because pattern matching (expansion of star, the question mark, and brackets) does not apply in this context. Note that the value of $asterisks is the literal string "$stars", not the string "*****", because the single quotes inhibit substitution.

In assignments, spaces are not re-interpreted after variable substitution, so that the following example results in $first and $second having the same value:

```
first='a string with embedded spaces'
second=$first
```

In accessing the values of variables, one may enclose the variable name (or digit, in positional parameters) in braces {...} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input:

```
a='This is a string'
echo "${a}ent test of variables."
```

Here, the echo command prints:

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for $aent and print:

```
ent test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user:

HOME Initialized by the login program to the name of the user's login directory, i.e., the directory that becomes the current directory upon completion of a login; cd without arguments uses $HOME as the directory to switch to. Using this variable helps keep full pathnames out of shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

IFS The variable that specifies which characters are internal field separators. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set IFS to include that delimiter.) The shell initially sets IFS to include the blank, tab, and newline characters.

MAIL The pathname of a file where your mail is deposited. If MAIL is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL must be set by the user and exported. (The presence of mail in the standard mail file is also announced at login, regardless of whether MAIL is set.)

PATH The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes PATH to the list :/bin:/usr/bin where a null argument appears in front of the first colon. A null anywhere in the path list represents the current directory. On some systems, a search of the current directory is not the default and the PATH variable is initialized instead to /bin:/usr/bin. If you wish to search your current directory last, rather than first, use:

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

Here, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, $HOME/bin) and cause it to be searched before the other three directories by using:

```
PATH=$HOME/bin::/bin:/usr/bin
```

The setting of PATH is normally done in your .profile file.

PS1 The variable that specifies what string is to be used as the primary prompt string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is "$ " (a dollar sign `$' followed by a blank).

PS2 The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2. The default value for this variable is "> " (a greater-than symbol `>' followed by a blank).

In general, you should be sure to **export** all of the above variables so that they are defined for all shells. Use **export** at the end of your .profile file. An example of an **export** statement follows:

```
export HOME IFS MAIL PATH PS1 PS1
```

7.6.3 Pre-Defined Special Variables

Several variables have special meanings; the following are set only by the shell:

$# Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, $# yields the number of the highest set positional parameter. Thus

```
sh cmd a b c
```

sets $# to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
        echo 'two or more args required'; exit
fi
```

$? Contains the exit status of the last command executed (also referred to as "return code," "exit code," or "value"). Its value is a decimal string. Most XENIX commands return zero to indicate

successful completion. The shell itself returns the current value of $? as its exit status.

$$ The process number of the current process. Because process numbers are unique among all existing processes, this 5-digit string is often used to generate unique names for temporary files. XENIX provides no mechanism for the automatic creation and deletion of temporary files; a file exists until it is explicitly removed. Temporary files are generally undesirable objects; the XENIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files in a directory used only for that purpose:

```
:       use current process id
:       to form unique temp file
temp=$HOME/temp/$$
ls > $temp
:       commands here, some of which use $temp
rm $temp
:       clean up at end
```

$! The process number of the last process run in the background (using the ampersand (&)). Again, this is a 5-digit string.

$- A string consisting of names of execution flags currently turned on in the shell. For example, $- might have the value "xv" if you are tracing your output.

7.7 The Shell State

The state of a given instance of the shell includes the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory.

The state of a shell may be altered in various ways. These include changing the working directory with the cd command, setting several flags, and by reading commands from the special file, .profile, in your login directory. The .profile file is read each time you log in to XENIX, It is normally used to execute special one-time-only commands and to set and export variables to all later shells.

7.7.1 Cd

The **cd** command changes the current directory to the one specified as its argument. This can and should be used to change to a convenient place in the directory structure. Note that **cd** is often placed within parentheses to cause a sub-shell to change to a different directory and execute some commands without affecting the original shell.

For example, the first sequence below copies the file /usr/bin/ls to /usr/you/bin/ls. The second changes directories first, and then performs a copy.

```
cp /usr/bin/ls /usr/you/bin/ls
(cd /usr/bin ; cp ls /usr/you/bin/ls)
```

Both command lines have the same effect.

7.7.2 The .profile File

When you log in, the shell is invoked to read your commands. The shell then proceeds to see if you have a file named .profile in your login directory. If so, commands are read and executed from it. Finally, the shell is ready to read commands from your standard input -- usually the terminal.

7.7.3 Execution Flags: set

The **set** command lets you alter the behavior of the shell by setting certain shell flags. In particular, the -x and -v flags may be useful when invoking the shell as a command from the terminal. Flags may be **set** by typing, for example:

```
set -xv
```

(to turn on both flags -x and -v). The same flags may be turned off by typing:

```
set +xv
```

These two flags have the following meaning:

-v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.

-x Commands and their arguments are printed as they are executed. (Shell control commands, such as for, while, etc., are not printed, however.) Note that -x causes a trace of only those commands that are actually executed, whereas -v prints each line of input until a syntax error is detected.

The set command is also used to set these and other flags within shell procedures.

7.8 A Command's Environment

All the variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its environment. This environment includes variables that the command inherits from its parent process and variables specified as keyword parameters on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell and all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally before the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
:       keycommand
echo $a $b
```

is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are not counted as arguments to the procedure and do not affect $#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment

that the procedure passes to its child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, type:

```
export
```

You will also get a list of variables that have been made readonly. To get a list of name-value pairs in the current environment, type:

```
set
```

7.9 Invoking the Shell

The shell is a command and may be invoked in the same way as any other command:

sh *proc* [*arg* ...]	A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated.
sh -v *proc* [*arg*...]	This is equivalent to putting "set -v" at the beginning of *proc*. Similarly, for the **-x**, **-e**, **-u**, and **-n** flags.
proc [*arg* ...]	If *proc* is marked executable, and is not a compiled, executable program, the effect is similar to that of: `sh proc args` An advantage of this form is that variables that have been exported in the shell will still be exported from *proc* when this form is used (because the shell only forks to read commands from *proc*). Thus any changes made within *proc* to the values of exported variables will be passed on to subsequent commands invoked from *proc*.

7.10 Passing Arguments to the Shell

When a command line is scanned, any character sequence of the form $n is replaced by the nth argument to the shell, counting the name of the shell procedure itself as $0. This notation permits direct reference to the procedure name and to as many as nine positional parameters. Additional arguments can be processed using the shift command or by using a **for** loop.

The **shift** command shifts arguments to the left; i.e., the value of $1 is thrown away, $2 replaces $1, $3 replaces $2, and so on. The highest-numbered positional parameter becomes unset ($0 is never shifted). For example, in the shell procedure ripple below, echo writes its arguments to the standard output; while is discussed later (it is a looping command); and lines that begin with a colon (:) are comments:

```
:       ripple command
while test $# != 0
do
        echo $1 $2 $3 $4 $5 $6 $7 $8 $9
        shift
done
```

If the procedure were invoked by

```
ripple a b c
```

it would print:

```
a b c
b c
c
```

The special shell variable "star" ($*) causes substitution of all positional parameters except $0. Thus, the echo line in the ripple example above could be written more compactly as:

```
echo $*
```

These two echo commands are not equivalent: the first prints at most nine positional parameters; the second prints all of the current positional parameters. The shell star variable ($*) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command:

```
nroff -man $*
```

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a newline or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via back quotes) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for internal field separators, that is, for any characters specified by IFS to break the command line into distinct arguments; explicit null arguments (specified by "" or '') are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then filename generation occurs, with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes, command lines are built inside a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command **eval** is available for this purpose. **Eval** takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output=' | wc -l'
eval $command $output
```

This segment of code results in the command line

```
who | wc -l
```

being executed.

The output of **eval** cannot be redirected. However, uses of **eval** can be nested, so that a command line can be evaluated several times.

7.11 Control Commands

The shell provides several commands that implement a variety of control structures useful in creating shell procedures. Before describing these structures, a few terms need to be defined.

A simple command is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, not to the command.

A command is a simple command or any of the shell control commands described below. A pipeline is a sequence of one or more commands separated by a vertical bar (|). In a pipeline, the standard output of each command but the last is connected (by a pipe) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is non-zero if the exit status of either the first or last process in the pipeline is non-zero.

A command list is a sequence of one or more pipelines separated by a semicolon (;), an ampersand (&), an "and-if" symbol (&&), or an "or-if" (||) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing <INTERRUPT> or <QUIT>. It is also immune to logouts with <CONTROL-D>. However, <CONTROL-D> will abort the command if you are operating over a dial-up line. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. In the above example without **nohup**, if you log out from a dial-up line while cc is still executing, cc will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of background processes without a compelling reason for doing so.

The and-if and or-if (&& and ||) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are lower than the ampersand (&) and the vertical bar (|)). In the command line

```
cmd1 || cmd2
```

the first command, cmd1, is executed and its exit status examined. Only if cmd1 fails (i.e., has a non-zero exit status) is cmd2 executed. Thus, this is a more terse notation for:

```
if      cmd1
        test $? != 0
then
        cmd2
fi
```

The and-if operator (&&) operator yields a complementary test. For example, in the following command line

```
cmd1 && cmd2
```

the second command is executed only if the first succeeds (and has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents:

```
{ nroff -mm text1; nroff -mm text2; } | lpr
```

Note that a space is usually needed after the left brace and before the right brace.

7.11.1 **Structured Conditional: if**

The shell provides structured conditional capability with the if command. The simplest if command has the following form:

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

The command list following the if is executed and if the last command in the list has a zero exit status, then the command list that follows then is executed. The word fi indicates the end of the if command.

To cause an alternative set of commands to be executed when there is a non-zero exit status, an else clause can be given with the following structure:

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

Multiple tests can be achieved in an if command by using the elif clause, although the case statement is better for large numbers of tests. For example:

```
if      test -f "$1"
:                       is $1 a file?
then    pr $1
elif    test -d "$1"
:                       else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows: if the value of the first positional parameter is a filename, then print that file; if not, then check to see if it is the name of a directory. If so, change to that directory and print all the files there. Otherwise, echo the error message.

The if command may be nested (but be sure to end each one with a fi). The newlines in the above examples of if may be replaced by semicolons.

The exit status of the if command is the exit status of the last command executed in any then clause or else clause. If no such command was executed, if returns a zero exit status.

7.11.2 Multiple Way Branch: case

A multiple way branch is provided by the case command. The basic format of case is:

```
case string in
        pattern ) command-list ;;
        ...
        pattern ) command-list ;;
esac
```

The shell tries to match string against each pattern in turn, using the same pattern-matching conventions as in filename generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the case and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a star (*) is the first pattern in a case, no other patterns are looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by vertical bars (|). For example:

```
case $i in
        *.c)    cc $i
                ;;
        *.h | *.sh)
                : do nothing
                ;;
        *)      echo "$i of unknown type"
                ;;
esac
```

In the above example, no action is taken for the second set of patterns because the null, colon (:) command is specified. The star (*) is used as a default pattern, because it matches any word.

The exit status of case is the exit status of the last command executed in the case command. If no commands were executed, then case has a zero exit status.

7.11.3 Conditional Looping: while and until

A **while** command has the general form:

```
while command-list
do
        command-list
done
```

The commands in the first command list are executed, and if the exit status of the last command in that list is zero, then the commands in the second list are executed. This sequence is repeated as long as the exit status of the first command list is zero. A loop can be executed as long as the first command list returns a non-zero exit status by replacing **while** with **until**.

Any newline in the above example may be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second command list. If no such command is executed, **while** (or **until**) has a zero exit status.

7.11.4 Looping Over a List: for

Often, one wishes to perform some set of operations for each file in a set of files, or execute some command once for each of several arguments. The **for** command can be used to accomplish this. The **for** command has the format

```
for variable in word-list
do
        command-list
done
```

where word-list is a list of strings separated by blanks. The commands in the command-list are executed once for each word in the word list. Variable takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following **for** loop causes each of the C source files xec.c, cmd.c, and word.c in the current directory to be "diffed" with a file of the same name in the directory /usr/src/cmd/sh:

```
for CFILE in xec cmd word
do      diff $CFILE.c /usr/src/cmd/sh/$CFILE.c
done
```

The first occurrence of CFILE immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

One can omit the "in word-list" part of a **for** command; this causes the current set of positional parameters to be used in place of word-list. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments. For example, assume that the following command is given the name "echo2":

```
for word
do echo $word$word
done
```

Then this command could be given execute status and executed as follows:

```
chmod +x echo2
echo2 ma pa bo fi yo no so ta
```

The output from this command would be:

```
mama
papa
bobo
fifi
yoyo
nono
soso
tata
```

7.11.5 Loop Control: break and continue

The **break** command can be used to terminate execution of a **while** or a **for** loop. **Continue** requests the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done.**

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done.** Exit from n levels is obtained by **break** n.

The **continue** command causes execution to resume at the nearest enclosing **for, while,** or **until** statement, i.e., the one that begins the innermost loop containing the **continue.** You can also specify an argument n to **continue** and execution will resume at the nth enclosing loop:

```
: This procedure is interactive.
: "Break" and "continue" commands are used
: to allow the user to control data entry.
while true
do      echo "Please enter data"
        read response
        case "$response" in
        "done") break
                : no more data
                ;;
        "")     continue
                ;;
        *)      : process the data here
                ;;
        esac
done
```

7.11.6 End-of-File and exit

When the shell reaches the end-of-file, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a <CONTROL-D> which is the same as logging out.

The exit command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing "`exit 0" at the end of the file.

7.11.7 Command Grouping: Parentheses and Braces

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a sub-shell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command line -- they can appear as literal parentheses only by being quoted. For example, if you type

```
garble(stuff)
```

the shell interprets this as four discrete words: "garble", "(", "stuff", and ")".

This sub-shell capability is useful when performing operations without affecting the values of variables in the current shell, or when temporarily changing directory and executing commands in the new directory without having to explicitly return to the current directory.

The current environment is passed to the sub-shell and variables that are exported in the current shell are also exported in the sub-shell. Thus

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n | lpr& ; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n | lpr&)
```

accomplish the same result: a copy of /usr/docs/otherdir/doc.n is sent to the line printer. However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. The shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) may also be used to group commands together. Both the left and the right brace are recognized only if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no sub-shell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

7.11.8 Input/Output Redirection and Control Commands

The shell normally does not fork when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input/output is specified explicitly for a control command, a separate process is spawned to execute that command. Thus, when if, while, until, case, and for are used in a pipeline consisting of more than one command, the shell forks and a sub-shell runs the control command. This has two implications:

1. The most noticeable implication is that any changes made to variables within the control command are not

effective once that control command finishes (this is similar to the effect of using parentheses to group commands).

2. The second implication is that control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

7.11.9 Transfer to Another File and Back: The Dot (.) Command

A command line of the form:

```
. proc
```

causes the shell to read commands from proc without spawning a new process. Changes made to variables in proc are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the .profile file with:

```
. .profile
```

7.11.10 Interrupt Handling: trap

A program may choose to catch an interrupt from the terminal, to ignore it completely, or to be terminated by it. Shell procedures can use the **trap** command to obtain the same effects.

```
trap arg signal-list
```

is the form of the **trap** command, where arg is a string to be interpreted as a command list and signal-list consists of one or more signal numbers as described in signal(2)). The commands in arg are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in the file errdirect when it is interrupted, thus giving the user information as to how much of the job was done:

```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
           : commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotes does something different:

```
(trap "echo `pwd` >errdirect" 2 3 15)
```

This prints the name of the directory from which the procedure was executed.

Signal 11 may never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the trap command as a signal generated by exiting from a shell. This occurs either with an exit command, or by "falling through" to the end of a procedure. If arg is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If arg is an explicit null string ('' or ""), then the signals in the signal list are ignored by the shell.

The trap command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
     : commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15 (kill) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotes are executed. The exit command must be included, or else the shell continues reading commands where it left off when the signal was received. The "trap 0" in the above procedure turns off the original trap on exits from the shell, so that the exit command does not reactivate the execution of the trap commands.

Sometimes the shell continues reading commands after executing trap commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands

typed at the terminal until an end-of-file (<CONTROL-D>) or an interrupt is received. An end-of-file causes the **read** command to return a non-zero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:

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

Several traps may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the shell implements the trap command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate trap commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned after the signal was received.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

7.12 Special Shell Commands

There are several special commands that are internal to the shell (some of which have already been mentioned). These commands should be used whenever possible, because they are, in general, faster and more efficient than other XENIX commands. The shell does not fork to execute these commands, so no additional processes are spawned. The trade-off for this efficiency is that redirection of input/output is not allowed for most of these special commands.

Several of the special commands have already been described because they affect the flow of control. They are dot (.), **break**, **continue**, **exit**, and **trap**. The **set** command is also a special command. Descriptions of the remaining special commands are given here:

:	The null command. This command does nothing and is used to insert commments in shell procedures. Its exit status is zero (true). Beware: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.
cd arg	Make arg the current directory. If arg is not a valid directory, or the user is not authorized to access it, a non-zero exit status is returned. Specifying **cd** with no arg is equivalent to typing "cd $HOME".
exec arg ...	If arg is a command, then the shell executes it without forking. No new process is created. Input/output redirection arguments are allowed on the command line. If only input/output redirection arguments appear, then the input/output of the shell itself is modified accordingly.
newgrp arg ...	The **newgrp** command is executed, replacing the shell. **Newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any variables that were exported will no longer be marked as

such.

read var ...	One line (up to a newline) is read from standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the last variable. The exit status of **read** is zero unless an end-of-file is read.
readonly var ...	The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all **exported** variables is given.
times	The accumulated user and system times for processes run from the current shell are printed.
umask nnn	The user file creation mask is set to nnn. If nnn is omitted, then the current value of the mask is printed.
wait	The shell waits for all currently active child processes to terminate. The exit status of **wait** is always zero.

7.13 Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the mode of the file to make it executable, thus permitting it to be invoked by

```
proc args
```

rather than

```
sh proc args
```

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for longer-lived ones. Here is the entire input needed to set up a simple procedure:

```
ed
a
nroff -T450 -man $*
.
w draft
q
chmod +x draft
```

It may then be invoked as

```
draft file1 file2
```

Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If draft were thus created in a directory whose name appears in the user's PATH variable, the user could change working directories and still invoke the draft command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the dot command (.) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer to write shell procedures instead of C programs. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.

2. A shell procedure has no corresponding object program that must be generated and maintained.

3. A shell procedure is easy to create on the fly, use a few times, and then remove.

4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named bin. This name is derived from the word "binary," and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more bin directories set up to hold common procedures. Some users

have their PATH variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it may become difficult to keep track of your environment and efficiency may suffer.

7.14 More About Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

-e This flag causes the shell to exit immediately if any command that it executes exits with a non-zero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.

-u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.

-t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line.

-n This is a "don't execute" flag. On occasion, one may want to check a procedure for syntax errors, but not execute the commands in the procedure. Using "set -nv" at the beginning of a file will accomplish this.

-k This flag causes all arguments of the form variable=value to be treated as keyword parameters. When this flag is not set, only such arguments that appear before the command name are treated as keyword parameters.

7.15 Supporting Commands and Features

Shell procedures can make use of any XENIX command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use.

7.15.1 Conditional Evaluation: test

The test command evaluates the expression specified by its arguments and, if the expression is true, test returns a zero exit status. Otherwise, a non-zero (false) exit status is returned. Test also returns a non-zero exit status if it has no arguments. Often it is convenient to use the test command as the first command in the command list following an if or a while. Shell variables used in test expressions should be enclosed in double quotes if there is any chance of their being null or not set.

On some XENIX systems, the square brackets may be used as an alias to test, for example

[expression]

has the same effect as:

test expression

Note that the spaces before and after the expression in brackets are essential.

The following is a partial list of the primaries that can be used to construct a conditional expression:

-r file	True if the named file exists and is readable by the user.
-w file	True if the named file exists and is writable by the user.
-x file	True if the named file exists and is executable by the user.
-s file	True if the named file exists and has a size greater than zero.
-d file	True if the named file is a directory.
-f file	True if the named file is an ordinary file.
-z sl	True if the length of string sl is zero.
-n sl	True if the length of the string sl is non-zero.
-t fildes	True if the open file whose file descriptor number is fildes is associated with a terminal device. If fildes is not specified,

file descriptor 1 is used by default.

s1 = s2	True if strings s1 and s2 are identical.
s1 != s2	True if strings s1 and s2 are not identical.
s1	True if s1 is not the null string.
n1 -eq n2	True if the integers n1 and n2 are algebraically equal; other algebraic comparisons are indicated by -ne, -gt, -ge, -lt, and -le.

These primaries may be combined with the following operators:

!	unary negation operator.
-a	binary logical and operator.
-o	binary logical or operator; it has lower precedence than the binary logical or operator (-a).
(expr)	parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all primaries, operators, filenames, etc. are separate arguments to **test**.

7.15.2 Simple Output: echo

The **echo** command is invoked as:

echo [arg ... **]**

Echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. Often, it is used to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command

```
ls
```

is often replaced by

```
echo *
```

because the latter is faster and prints fewer lines of output.

The -n switch to **echo** removes the newline from the end of the echoed line. The following two commands prompt for input and then allow typing on the same line as the prompt:

```
echo -n 'enter name:'
read name
```

7.15.3 Expression Evaluation: expr

The **expr** command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; **expr** can be used inside grave accents to set a variable. Typical examples are:

```
:       increment $A
A=`expr $a + 1`
:       put third through last characters of
:       $1 into substring
substring=`expr "$1" : '..\(.*\) ' `
:       obtain length of $1
c=`expr "$1" :  '.* ' `
```

The most common uses of expr are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

7.15.4 True and False

The **true** and **false** commands perform the obvious functions of exiting with zero and non-zero exit status, respectively. The **true** command is often used to implement an unconditional loop.

7.15.5 In-Line Input Documents

Upon seeing a command line of the form

```
command << eofstring
```

where eofstring is any arbitrary string, the shell will take the subsequent lines as the standard input of command until a line is read consisting only of eofstring.

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, one may quote any character of eofstring:

```
command << \eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file.

7.15.6 Input/Output Redirection Using File Descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the write(2) system call. The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By typing

```
fd1>&fd2
```

where fd1 and fd2 are valid file descriptors, one can direct output that would normally be associated with file descriptor fd1 onto the file associated with fd2. The default value for fd1 and fd2 is 1. If, at run time, no file is associated with fd2, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing:

```
command 2>&1
```

If you wanted to redirect both standard output and standard error output to the same file, you would type:

```
command 1>file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with file; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to file, because at the time of the error output redirection, file descriptor 1 still would

have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard input. You could type

```
fda<&fdb
```

to cause both file descriptors fda and fdb to be associated with the same input file. If fda or fdb is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

7.15.7 Conditional Substitution

Normally, the shell replaces occurrences of $variable by the string value assigned to variable, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in anyone of the following ways:

```
A=
bcd=""
efg=''
set '' ""
```

The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. Parameter as used below refers to either a digit or a variable name.

${variable:-string} If variable is set and is non-null, then substitute the value $variable in place of this expression. Otherwise, replace the expression with string. Note that the value of variable is not changed by the evaluation of this expression.

${variable:=string} If variable is set and is non-null, then substitute the value $variable in place of this expression. Otherwise, set variable to string, and then substitute the value

$variable in place of this expression. Positional parameters may not be assigned values in this fashion.

${variable:?string} If variable is set and is non-null, then substitute the value of variable for the expression. Otherwise, print a message of the form

variable: string

and exit from the current shell. (If the shell is the login shell, it is not exited.) If string is omitted in this form, then the message

variable: parameter null or not set

is printed instead.

${variable:+string} If variable is set and is non-null, then substitute string for this expression. Otherwise, substitute the null string. Note that the value of the variable is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

```
PATH=${PATH:-':/bin:/usr/bin'}
```

   This says, if PATH has ever been set and is not null, then keep its current value; otherwise, set it to the string ":/bin:/usr/bin".

2. This example automatically assigns the HOME variable a value:

```
cd ${HOME:='/usr/gas'}
```

   If HOME is set, and is not null, then change directory to it. Otherwise set HOME to the given value and

change directory to it.

7.15.8 **Invocation** Flags

There are four flags that may be specified on the command line when invoking the shell. These flags may not be turned on with the **set** command:

-i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is interactive. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.

-s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. The shell you get upon logging into the system has the -s flag turned on.

-c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored. Double quotes should be used to enclose a multi-word string, in order to allow for variable substitution.

7.16 Effective and Efficient Shell Programming

This section outlines strategies for writing efficient shell procedures, ones that do not unreasonably waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the few important parts. The user should become familiar with the time command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

7.16.1 Number of Processes Generated

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built-in:

```
break   case    cd        continue   eval
exec    exit    export    for        if
newgrp  read    readonly  set        shift
test    times   trap      umask      until
wait    while   .         :          {}
```

Parentheses, (), are built into the shell, but commands enclosed within them are executed as a child process, i.e.,

the shell does a **fork**, but no **exec**. Any command not in the above list requires both **fork** and **exec**.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

processes = (k*n) + c

where k and c are constants, and n may be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of k, sometimes to zero.

As an example, here is an analysis of a procedure named split, whose text is given below:

```
:       split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
                : read stdin into temp file
                : save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
                : lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                : lines with only numbers onto $2
total="`wc -l < temp$$`"
end1="`wc -l < $1`"
end2="`wc -l < $2`"
lost="`expr $total - \($end1 - $start1\) - \($end2 - $start2\)`"
echo "$total read, $lost thrown away"
```

For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr**. One additional **echo** is executed at the end. If n is the number of lines of input, the number of processes is 2*n+1. On the other hand, the number of processes in the following (equivalent) procedure is 12, regardless of the number of lines of input:

Some types of procedures should not be written using the shell. For example, if one or more processes are generated

for each character in some file, it is a good indication that the procedure should be rewritten in C.

Shell procedures should not be used to scan or build files a character at a time.

7.16.2 Number of Data Bytes Accessed

It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the shrinkers first when the order is irrelevant. For instance, the second of the following examples is likely to be faster:

```
sort file | grep pattern
grep pattern file | sort
```

7.16.3 Directory Searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd** can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system):

```
time sh -c 'ls -l /usr/bin/* >/dev/null'
time sh -c 'cd /usr/bin; ls -l * >/dev/null'
```

7.16.4 Directory-Search Order and the PATH Variable

The PATH variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead that occurs in a subtle, but avoidable, way.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current PATH variable. As an example, consider the effect of invoking **nroff** (i.e., /usr/bin/nroff) when the value of PATH is ":/bin:/usr/bin". The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to PATH can increase this number significantly.

The vast majority of command executions are of commands found in /bin and, to a somewhat lesser extent, in /usr/bin. Careless PATH setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best (but only with respect to the efficiency of command searches):

```
:/usr/john/bin:/usr/lbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/lbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/lbin
/bin::/usr/bin:/usr/john/bin:/usr/lbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in /bin and /usr/bin.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the PATH variable inside the procedure so that the fewest possible directories are searched in an optimum order.

7.16.5 Good Ways to Set Up Directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small file; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink.

7.17 Shell Procedure Examples

The power of the XENIX shell command language is most readily seen by examining how XENIX's many labor-saving utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called "scripts"). Note the use of the null command (:) for introducing comments into shell procedures.

All of the following procedures could conceivably be entered at your keyboard. However, it is intended that each procedure be placed in file, given execute permission with the **chmod** command, and then executed like any other XENIX command.

BINUNIQ

```
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both /bin and /usr/bin. It is done because files in /bin will "override" those in /usr/bin during most searches and duplicates need to be weeded out. If the /usr/bin file is obsolete, then space is being wasted; if the /bin file is outdated by a corresponding entry in /usr/bin then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of "sort | uniq" to find matches and/or duplications.

COPYPAIRS

```
:       Usage: copypairs file1 file2 ...
:       Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
        cp $1 $2
        shift; shift
done
if test "$1" != ""
then echo "$0: odd number of arguments"
fi
```

This procedure illustrates the use of a while loop to process a list of positional parameters that are somehow related to one another. Here a while loop is much better than a for loop, because you can adjust the positional parameters with the shift command to handle related arguments.

COPYTO

```
:       Usage: copyto dir file ...
:       Copies argument files to "dir",
:       making sure that at least
:       two arguments exist, that "dir" is a directory,
:       and that each additional argument
:       is a readable file.
if test $# -lt 2
then    echo "$0: usage: copyto directory file ..."
elif test ! -d $1
then    echo "$0: $1 is not a directory";
else    dir=$1; shift
        for eachfile
        do      cp $eachfile $dir
        done
fi
```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first; the original **$1** is shifted off.

DISTINCTl

```
:       Usage: distinctl
:       Reads standard input and reports list of
:       alphanumeric strings that differ only in case,
:       giving lowercase form of each.
tr -cs 'A-Za-z0-9' '\012'|sort -u | \
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It may not be immediately obvious how this command works. You may wish to consult tr(1), sort(1), and uniq(1) if you are completely unfamiliar with these commands. The tr command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The sort command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lower case, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. The first line is equivalent to the last two lines, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic **distinctl** with such a step-by-step process, using a file of test data containing:

```
ABC:DEF/DEF
ABCl ABC
Abc abc
```

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

DRAFT

```
:       Usage: draft file(s)
:       Print man pages for Diablo printer.
for i in $*
        do nroff -T450 -man $i | lpr
        done
```

Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
:       Usage: edfind file arg
:       Finds the last occurrence in "file" of a line
:       whose beginning matches "arg", then prints
:       3 lines (the one before, the line itself,
:       and the one after)
ed - $1 <<!
?^$2?
-,+p
!
```

This illustrates the practice of using editor (ed) in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
:       Usage: edlast file
:       Prints the last line of file,
:       then deletes that line.
ed - $1 <<\!
        $p
        $d
        w
        q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation (!). Variable substitution is prohibited because of the backslash.

FILCOM

```
if (cmp -s $1 $2 = 0)
  then
    echo Files are identical
  else
    diff $1 $2
fi
```

This procedure compares two files ($1 and $2) and if they are identical reports the fact that the files are indeed the same. If they are different, it lists the differences between them.

FSPLIT

```
:       Usage: fsplit file1 file2
:       Reads standard input and divides it into 3 parts:
:       appends any line containing at least one letter
:       to file1, any line containing digits but no
:       letters to file2, and throws the rest away.
count=0 gone=0
while read next
do
        count="`expr $count + 1`"
        case "$next"  in
        *[A-Za-z]*)
                echo  "$next"  >> $1 ;;
        *[0-9]*)
                echo  "$next"  >> $2 ;;
        *)
                gone="`expr $gone + 1`"
        esac
done
echo "$count lines read, $gone thrown away"
```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when read encounters an end-of-file. Note the use of the expr command.

Don't use the shell to read a line at a time unless you must -- it can be an extremely slow process.

GDIF

```
PATH=/bin:/usr/bin
: ${STD=}

if test "$1" = "-h"; then
        diff="diff -hb"
        shift
else
        diff="diff -b"
fi
for i
do
        if test -f $i
        then    if test ! -f $STD/$i
                then    echo "New file: $i"
                        if test `tail -41 $i | wc -l |\
                                tr -d ' ' ` -lt 41
                        then cat $i
                        fi
                        echo ""
                elif cmp -s $i $STD/$i; then
                        :
                else    echo "$i -- $STD/$i"
                        $diff $i $STD/$i
                        echo ""
                fi
        fi
done
```

This procedure can be used to find the differences between one set of files and another set which must be in a similar directory hierarchy. The second set of filenames is constructed by prepending a string to each name in the first set. The string to be used may be exported to the procedure in the variable STD; by default it is a slash (/). The procedure will also indicate if new files have appeared in the first set of names. If a new file is relatively short, it will be printed. If a -h flag is given, the diffh program will be used to list the differences.

INITVARS

```
:       Usage: . initvars
:       Uses carriage returns for "no change."
echo -n 'initializations? '
read response
if test "$response" = y
then    echo -n 'HOME='; read temp
        HOME=${temp:-$HOME}
        echo -n "PATH="; read temp
        PATH=${temp:-$PATH}
        echo -n "TERM="; read temp
        TERM=${temp:-$TERM}
fi
```

This shell procedure would be invoked by a user at the terminal, or as part of a .profile file. The assignments are effective even when the procedure is finished, because the "dot" command is used to invoke it. To better understand the dot command, invoke initvars as indicated above and check the values of HOME, PATH, and TERM. Then make initvars executable, type initvars, assigning different values to the three variables, and check again the values of these three shell variables after initvars terminates.

LISTFIELDS

```
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a semicolon on a newline. Thus, if given the following input

```
joe newman: 13509 NE 78th St: Redmond, Wa 98062
```

listfields will produce this:

```
joe newman
 13509 NE 78th St
 Redmond, Wa 98062
```

Note the use of the tr command to transpose colons to line feeds.

MERGE

```
:       Usage:  merge srcl src2 [ dest ]
:       Merges two files, every other line.
:       The first argument starts off the merge,
:       excess lines of the longer file are appended to
:       the end of the resultant file.
exec 4<$1 5<$2
: default dest. file below is named $1.m
dest=${3-$1.m}
while true
do
                : Alternate reading from the files;
                : The variable 'more' represents the file
                : descriptor of the longer file.
        line <&4 >>$dest || { more=5; break ;}
        line <&5 >>$dest || { more=4; break ;}
done
                : Delete the last line of dest
                : file, because it is blank.
ed - $dest <<!
        \$d
        w
        q
!
while line <&$more >> $dest
do :; done
                : Read the remainder of the longer file.
                : The body of the "while" loop
                : does nothing; the work of the loop
                : is done in the command list following
                : the "while".
```

This command illustrates a technique for reading sequential lines from a file or files without creating any sub-shells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used srcl and src2 explicitly rather than the associated file descriptors, this procedure would never terminate, because the first line of each file would be read over and over again.

The Shell

MKFILES

```
:       Usage: mkfiles pref [quantity]
:       Makes "quantity" files, named pref1, pref2, ...
:       Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
        > $1$i
        i="`expr $i + 1`"
done
```

The mkfiles procedure uses input/output redirection to create zero-length files. The expr command is used for counting iterations of the while loop.

NULL

```
:       Usage: null files
:       Create each of the named files as an empty file.
for eachfile
do
        >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
:       Usage: phone initials ...
:       Prints the phone numbers of the
:       people with the given initials.
echo 'inits     ext     home'
grep "^$1" <<!
jfk     1234    999-2345
lbj     2234    583-2245
hst     3342    988-1010
jqa     4567    555-1234
!
```

This procedure is an example of using an in-line input script to maintain a small data base.

STATLOG

```
(systat 600 &
 while sleep 3500; do
    date
 done) >file&
```

To run two programs simultaneously, and intermingle their output, you can do:

```
(prog1 &
prog2) >outfile
```

Thus, the above statlog procedure performs a systat every 10 minutes. Every almost-hour, the date is interjected into the file. In order to get "delta" figures, you don't want to rerun systat each time, but have him sleep, thus the 600.

SUBMIT

```
: "Submit a job for later execution, redirect outputs,
:  send mail when done"

if test $# = 2; then
        if test ! -r $2
        then    echo "$0: cant find $2"
                exit 1
        fi
        if test "$1" = now
        then    i=`date | \
                sed -e 's/^.*\(..\):\(..\):.*$/\1\2/'`
                i=`expr $i + 2`
                echo "Will start at $i."
        else    i="$1"
        fi

        usr=`who am i | awk '{print $1}'`
        at "$i" <<!
                ( echo -n "Started at:  "; date
                  sh $2
                  echo -n "Finished at: ";date
                  ) 1>$2out 2>&1
                echo "`pwd`/$2 finished." | mail $usr
!
elif    test $# = 1; then
        $0 2100 $1
else
        echo usage: $0 [ time ] shfile
fi
```

This procedure lets you submit a batch job that saves the results and notifies you that the job is done. It does this by placing a process in the background, writing results to a file, and then sending mail to the user.

TEXTFILE

```
if test "#$1" = #-s
then
:       Return condition code
        shift
        if test -z "`$0 $*`"; then
                exit 1
        else
                exit 0
        fi
fi

if test $# -lt 1
then    echo "$0: Usage: $0 [ -s ] file ..." 1>&2
        exit 0
fi

file $* | fgrep ' text' | sed 's/:      .*//'
```

To determine which files in a directory contain only textual information, textfile filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr `textfile *` | lpr
```

This procedure also uses a -s flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```
:       Usage: writemail message user
:       If user is logged in,
:       writes message to terminal;
:       otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by $1 is piped to both the **write** command and, if **write** fails, to the **mail** command.

7.18 Shell Grammar

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

simple-command: item
                simple-command item

command:        simple-command
                ( command-list )
                { command-list }

                for name
                        do command-list
                        done

                for name in word
                        do command-list
                        done

                while command-list
                        do command-list
                        done

                until command-list
                        do command-list
                        done

                case word in
                        case-part
                esac

                if command-list
                then command-list
                else-part
                fi

pipeline:               command
                pipeline | command

andor:          pipeline
                andor && pipeline
                andor || pipeline

command-list:   andor
                command-list ;
                command-list &
                command-list ; andor
                command-list & andor
```

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

file:           word
                & digit
                & -

case-part:      pattern ) command-list ;;

pattern:                word
                pattern | word

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

empty:

word:           a sequence of non-blank characters

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

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

Meta-characters and Reserved Words

a. syntactic

|	pipe symbol
&&	and-if symbol
||	or-if symbol
;	command separator
;;	case delimiter
&	background commands
( )	command grouping
<	input redirection
<<	input from a here document
>	output creation
<	output append

b. patterns

*	match any character(s) including none
?	match any single character
[...]	match any of enclosed characters

c. substitution

${...}	substitute shell variable
`...`	substitute command output

d. quoting

\	quote next character as literal with no special meaning

'...' quote enclosed characters excepting the back quote (')

"..." quote enclosed characters excepting: $ ` \ "

e. reserved words

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

CHAPTER 8

SED

CONTENTS

8.1 Introduction

Sed is a context editor designed to be used in three cases:

1. To edit files too large for comfortable interactive editing.

2. To edit files of any size when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.

3. To perform multiple "global" editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in memory at one time in Sed, and no temporary files are used, the effective size of a file that can be edited is virtually unlimited. Sed, by its nature, is non-interactive, and cannot be used at the terminal like the editors ed, and vi.

Complicated editing scripts can be created separately and given to sed as a command file. For complex edits, this saves considerable typing and its attendant errors. Sed, running from a command file, is much more efficient than any known interactive editor, even if that editor can be driven by a pre-written script.

The principal disadvantages of sed, compared to an interactive editor, are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a descendant of the XENIX line editor, ed(1). Because of the differences between interactive and non-interactive operation, considerable changes have been made between the two; even confirmed users of ed will frequently be surprised, if they rashly use sed without reading this chapter. The most striking family resemblance between the two editors is in the class of patterns (regular expressions) they recognize; the code for matching patterns is copied almost verbatim from the code for ed.

8.2 Overall Operation

Sed copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line.

The general format of an editing command is:

[address1,address2] [function] [arguments]

One or both addresses may be omitted. Any number of blanks or tabs may separate the addresses from the function, but the function must be present. The arguments may be required or optional, according to the function given. Tab characters and spaces at the beginning of lines are ignored.

8.3 Command Line Flags

Three flags are recognized on the command line:

-n No copy of input to output. This option suppresses the automatic copying of lines to the output. The only lines copied are those specified by the print functions (p or P) or by p flags.

-e Tells sed to take the next argument as an editing command. This next argument should be quoted with single quotation marks (') to prevent expansion of metacharacters by the shell. If only one -e argument is given and if there are no -f flags, then the -e flag may be omitted and the argument given as the first one on the command line.

-f Tells sed to take the next argument as a filename. The file should contain editing commands, one to a line.

Some sample invocations are given below:

```
sed -f sed.script <input >output
sed /s/one/two/g <input >output
sed -e s/one/two/g <input >output
sed -n -e /s/one/two/p <input >output
```

The first of these examples reads commands from the file sed.script. The second and third examples perform an identical operation: both take the first argument as a sed substitute command. The fourth example has the same effect as examples two and three, except that input lines are not copied to output.

Sed

8.4 Order of Application of Editing Commands

Commands are applied in the order that they are given on the command line or in the file specified by the **-f** flag. Each line may undergo several transformations before being output, depending on the number of commands to be executed. The output from each command is always a single line. This output line is the input to the next command to be executed. Thus, sed, is said to be a "line-at-a-time" editor as illustrated in Figure 8-1.

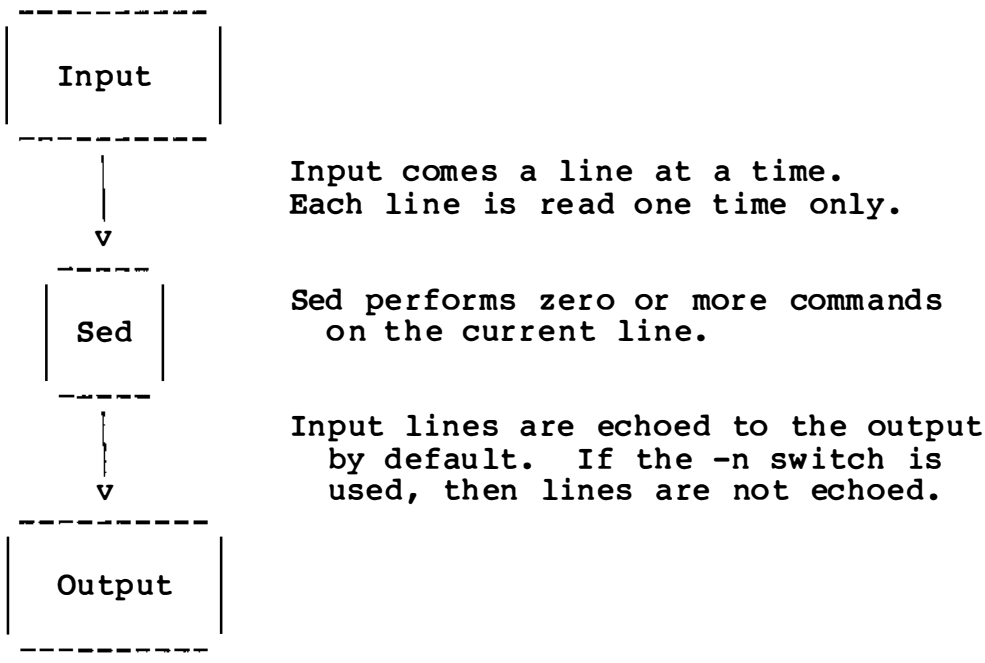


Figure 8-1. Line at a Time Processing

Note that the default linear order of application of editing commands can be changed by the flow-of-control commands, **t** and **b**.

8.5 The Pattern Space

The range of pattern matches is called the pattern space. Ordinarily the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the **N** command.

Except where otherwise noted, all examples in this chapter assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

The following is an an example of a command:

```
2q
```

This quits after copying the first two lines of the input. The output is:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

8.6 Addresses: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ({ and }).

8.6.1 Line-Number Addresses

A linenumber is a decimal integer. As each line is read from the input, a linenumber counter is incremented. A linenumber address matches and selects the input line equal to that line number in the input file. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened.

As a special case, the dollar sign ($) matches the last line of the last input file.

8.6.2 Context Addresses

A context address is a pattern (regular expression) enclosed in slashes (/). The regular expressions recognized by sed are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression, and matches that character.

2. A circumflex (^) at the beginning of a regular expression matches the beginning of a line.

3. A dollar-sign ($) at the end of a regular expression matches the end of a line.

4. The characters (\n) match an embedded newline character, but not the newline at the end of the pattern space.

5. A period (.) matches any character except the terminal newline of the pattern space.

6. A regular expression followed by an asterisk (*) matches any number (including 0) of adjacent occurrences of the regular expression it follows.

7. A string of characters in square brackets ([and]) matches any character in the string, and no others. If, however, the first character of the string is a circumflex (^), the regular expression matches any character except the characters in the string and the terminal newline of the pattern space.

8. A concatenation of regular expressions is a regular expression that matches the concatenation of strings matched by the components of the regular expression.

9. A regular expression between the sequences "\(" and "\)" is identical to the unadorned regular expression, but has side-effects which are described under the s command and 10, immediately below.

10. The expression \d means the same string of characters matched by an expression enclosed in "\(" and "\)" earlier in the same pattern. Here d is a single digit. The string specified is that beginning with the dth occurrence of "\(" counting from the left. For example, the expression "^\(.*\)\1" matches a line beginning with two repeated occurrences of the same string.

11. The null regular expression standing alone (for example, "//") is equivalent to the last regular expression searched for.

To use one of the special characters (^ $. * [ ] \ /) as a literal (to match an occurrence of itself in the input), precede or "escape" the special character with a backslash (\), as in "\$" or "*".

If a context address is to "match" the input, then the whole pattern within the address must match some portion of the pattern space.

8.6.3 Command Address Arguments

A command may have 0, 1, or 2 preceding address arguments, as described below:

1. If a command has no addresses, it is applied to every line in the input.

2. If a command has one address, it is applied to all lines which match that address.

3. If a command has two addresses (separated by a comma), it is applied to the first line that matches the first address, and to all subsequent lines up to and and including the first line matching the second address. This process is repeated, so that an attempt is made on subsequent lines to again match the first address.

4. An error occurs if a command has more addresses than the maximum allowed

   For example, examine the input text:

   ```
   In Xanadu did Kubla Khan
   A stately pleasure dome decree:
   Where Alph, the sacred river, ran
   Through caverns measureless to man
   Down to a sunless sea.
   ```

   The following addresses match lines of this text as noted:

/an/	Matches lines 1, 3, 4 in our sample text
/an.*an/	Matches line 1
/^an/	Matches no lines
/./	Matches all lines
/\./	Matches line 5
/r*an/	Matches lines 1, 3, and 4
/\(an\).*\1/	Matches line 1

8.7 Functions

All functions are named by a single character. In the following descriptions, the single character function name is given followed by possible arguments. Next, a description of what each function does is followed by the maximum number of allowable addresses that may precede the command given. This number is given in parentheses.

8.7.1 Whole-Line Oriented Functions

The following commands operate on whole lines; they do not affect intra-line contents:

d -- Delete lines (2)
The d function deletes from the input (i.e., does not write to the output) all those lines matched by its addresses. A side effect is that no further commands are attempted on the deleted line. As soon as the d function is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.

n -- Next line (2)
The n function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the n command.

a\ text -- Append lines (1)
The **a** function causes the text argument to be written to the output after the line matched by its address. The a command is inherently multi-line: the **a** itself must appear at the end of a line and must be followed by a backslash. The appended text may contain any number of lines. However, to preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character (\) immediately preceding the newline. The text argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash). Remember that text does not appear on the same line as the **a** command itself, and that the a command must always be followed by a backslash to have any effect, as in

```
a\
one line of appended text\
the second and last line with no backslash.
```

Once an a function is successfully executed, text is written to the output regardless of what later commands do to the line that triggered it. The triggering line may be deleted entirely, but text is still written to the output.

The text is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the linenumber counter.

i\ text -- Insert lines (1)

The i function is similar to the a function, except that text is written to the output before the matched line. All other comments about the a function apply to the i function as well.

c\ text -- Change lines (2)

The c function deletes the lines selected by its address(es), and replaces them with the lines in text. Like a and i, c must be followed by a newline hidden by a backslash; interior new lines in text must be hidden by backslashes.

The c command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of text is written to the output, not one copy per line deleted. As with a and i, text is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a c function, no further commands are attempted on it.

If text is appended after a line by either the a or r function, and the line is subsequently changed, the text inserted by the c function is placed before the text of the a or r function.

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in sed commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

As an example, examine the following list of editing commands:

```
n
a\
XXXX
d
```

These commands, applied to our standard input, produce:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n               n
i\              c\
XXXX            XXXX
d
```

8.7.2 The Substitute Function

The substitute function changes parts of lines selected by a context search within the line. Its syntax is below:

s/pattern/replacement/flags

The s function replaces part of a line (selected by pattern) with replacement. It can best be read as substitute for pattern, replacement. The pattern argument contains a pattern, exactly like the patterns in addresses. The only difference between pattern and a context address is that the context address must be delimited by slash (/) characters; pattern may be delimited by any character other than space or newline.

By default, only the first string matched by pattern is replaced, but see the g flag below.

The replacement argument begins immediately after the second delimiting character of pattern, and must be followed immediately by another instance of the delimiting character. (Thus, there are exactly three instances of the delimiting character.)

The replacement is not a pattern, and the characters which are special in patterns do not have special meanings in replacement. Instead, other characters are special:

& The ampersand is replaced by the string matched by pattern

\d The single digit d is replaced by the dth substring matched by parts of pattern enclosed in "\(" and "\)". If nested substrings occur in pattern, the dth is determined by counting opening delimiters "\(".

As in patterns, special characters may be made literal by preceding them with backslash (\).

The flags argument may contain the following flags:

g Substitute replacement for all (non-overlapping) instances of pattern in the line. After a successful substitution, the scan for the next instance of pattern begins just after the end of the inserted characters; characters put into the line from replacement are not re-scanned.

p Print the line if a successful replacement was done. The p flag causes the line to be written to the output if and only if a substitution was actually made by the s function. Note that if several s functions, each followed by a p flag, successfully substitute in the same input line, multiple copies of the line will be written to the output; one for each successful substitution.

w filename
Write the line to a file if a successful replacement was done. The w flag causes lines which are actually substituted by the s function to be written to a file named by filename. If filename exists before sed is run, it is overwritten; if not, the filename is created. A single space must separate w and filename. The possibilities of multiple, somewhat different copies of one input line being written are the same as for p. A maximum of ten different filenames may be mentioned after w flags and w functions (see below), combined.

For example, study the following command

```
s/to/by/w changes
```

applied to our standard input:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

This produces the following output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

It also writes the following output to the file changes:

```
Through caverns measureless by man
Down by a sunless sea.
```

If the no-copy option is in effect, the command

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the **g** flag, the command

```
/X/s/an/AN/p
```

produces (assuming no-copy mode):

```
In XANadu did Kubla Khan
```

and the command

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubla KhAN
```

8.7.3 Input/Output Functions

p -- Print (2)

The print function writes the addressed lines to the standard output file. They are written at the time the p function is encountered, regardless of what succeeding editing commands may do to the lines.

w filename -- Write to filename (2)

The write function writes the addressed lines to the file named by filename. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them. Exactly one space must separate the w and filename. A maximum of ten different files may be mentioned both for write functions and for w flags after s functions.

r filename -- Read the contents of a file (1)

The read function reads the contents of filename, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If r and a functions are executed on the same line, the text from the a functions and the r functions is written to the output in the order that the functions are executed.

Exactly one space must separate the r and filename. If a file mentioned by an r function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

Note: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files are mentioned in w functions or flags. This number is reduced by one if any r functions are present, since only one read file is open at one time.

For example, assume that the file notel has the following contents:

```
Note: Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
```

Then the following command:

```
/Kubla/r notel
```

produces:

```
In Xanadu did Kubla Khan
    Note:  Kubla Khan (more properly Kublai Khan; 1216-1294)
    was the grandson and most eminent successor of Genghiz
    (Chingiz) Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

8.7.4 Multiple Input-Line Functions

Three functions, all spelled with capital letters, deal with multiple line pattern spaces containing embedded newlines. They are intended to provide pattern matches across lines in the input.

N -- Next line (2)
The next input line is appended to the current line in the pattern space; the two input lines are separated by an embedded newline. Pattern matches may extend across the embedded newline(s).

D -- Delete first part of the pattern space (2)
Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from the beginning.

P -- Print first part of the pattern space (2)
Print up to and including the first newline in the pattern space.

The **P** and **D** functions are equivalent to their lowercase counterparts if there are no embedded newlines in the pattern space.

8.7.5 Hold and Get Functions

Five functions save and retrieve part of the input for possible later use:

h -- hold pattern space (2)
The **h** function copies the contents of the pattern space into a hold area, destroying the previous contents of the hold area.

H -- append pattern space to hold area (2)
The **H** function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

g -- get contents of hold area (2)
The **g** function copies the contents of the hold area into the pattern space, destroying the previous contents of the pattern space.

G -- Get contents of hold area (2)
The **G** function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

x -- Exchange (2)
The exchange command interchanges the contents of the pattern space and the hold area.

For example, the commands

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

Sed

8.7.6 Flow-of-Control Functions

These functions do no editing on the input lines, but rather control the application of functions to the lines selected by the address part.

! -- Don't (2)
The Don't command causes the next command (written on the same line), to be applied to all and only those input lines not selected by the address part.

{ and } -- Grouping (2)
The grouping command, {, causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line. The group of commands is terminated by a matching } standing on a line by itself. Groups can be nested.

:label -- Place a label (0)
The label function marks a place in the list of editing commands which may be referred to by b and t functions. The label may be any sequence of eight or fewer characters. If two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

blabel -- Branch to label (2)
The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same label was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A b function with no label is taken to be a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

tlabel -- Test substitutions (2)
The t function tests whether any successful

substitutions have been made on the current input line. If so, it branches to label; if not, the **t** function does nothing. The flag which indicates that a successful substitution has been executed is reset by reading a new input line, or executing a **t** function.

8.7.7 Miscellaneous Functions

= -- Equals (1)
The = function writes to the standard output the line number of the line matched by its address.

q -- Quit (1)
The **q** function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

8.8 Command Summary

a\ text	Append line to pattern space.
blabel	Branch to label.
c\ text	Change pattern space, replacing with text.
d	Delete pattern space.
D	Delete first part of pattern space.
g	Get contents of hold area.
G	Get first line of hold area.
h	Put pattern space in hold area.
H	Append pattern space to hold area.
i\ text	Insert lines.
n	Make next line pattern space.
N	Append next line to pattern space.
p	Print the pattern space.
P	Print first part of the pattern space.
q	Quit.
r filename	Read in the contents of a filename.
s/pattern/replacement/	Substitute pattern with replacement.
tlabel	Test substitutions.
w filename	Write to filename.
x	Exchange hold area with pattern space.
{ and }	Group statements.
addresses!	Don't match these addresses.
:label	Place label.

= Print line number.

CHAPTER 9

BC: A CALCULATOR

CONTENTS

9.1 Introduction

BC is a program for doing arbitrary precision arithmetic. BC is both a language and a compiler. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers. Although you can write substantial programs with BC, it is also a useful interactive tool for performing calculator-like computations. The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution. The execution speed of BC is quite fast. For example, two 500-digit numbers can be multiplied to give a 1000-digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of BC are to perform:

- Computation with large integers
- Computations accurate to many decimal places
- Conversions of numbers from one base to another base

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal simply by setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of XENIX .

The syntax of BC has been deliberately selected to agree substantially with the C language. Those who are familiar with C will find few surprises in this language.

9.2 Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type

```
142857 + 285714
```

the program responds immediately with the line:

```
428571
```

The following operators can all be used:

```
-   *   /   %   ^
```

They indicate subtraction, multiplication, division, modulo (remaindering), and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed with a minus sign to indicate that it is to be negated (this is the "unary" minus sign). For example, the expression

```
7+-3
```

is interpreted to mean that -3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in FORTRAN, with exponentiation (^) having the greatest binding power, then multiplication (*), division (/), and modulo (%), and, finally, addition (+), and subtraction (-). The contents of parentheses are evaluated before expressions inside the parentheses. All of the above operators are performed from left to right, except exponentiation, which is performed from right to left. The two expressions

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions:

```
a*b*c  and  (a*b)*c
```

BC shares with FORTRAN and C the convention that a/b*c is equivalent to (a/b)*c. Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named `x". When, as in this case, the outermost operator is the assignment operator (=), then the assignment is performed but the result is not printed. There are twenty-six available named storage registers.

There is a built-in square root function whose result is truncated to an integer (See also Section 9.4, "Scaling"). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

9.3 Bases

There are special internal quantities in BC, called ibase and obase. Ibase is initially set to 10, and determines the base used for interpreting numbers that are read by BC. For example, the lines

```
ibase = 8
11
```

produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. However, beware of trying to change the input base back to decimal by typing:

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement has no effect. For those who deal in hexadecimal notation, the characters A-F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10-15, respectively. The statement

```
ibase = A
```

changes you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted, but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The value of obase is initially set to 10 decimal. Obase is used as the base for output numbers. The lines

```
obase = 16
1000
```

produce the output line

```
3E8
```

which is to be interpreted as a three-digit hexadecimal number. Very large output bases are permitted. For example, large numbers can be output in groups of five digits by setting obase to 100000. Strange output bases, such as negative bases, and 1 and 0, are handled correctly.

Very large numbers are split across lines with seventy characters per line. A split line that continues on the next line ends with a backslash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

Remember that ibase and obase do not affect the course of internal computation or the evaluation of expressions; they only affect input and output conversion.

9.4 Scaling

A third special internal quantity called scale is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

Addition and subtraction
: The scale of the result is the larger of the scales of the two operands. There is never any truncation of the result.

Multiplication
: The scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands, and subject to those two restrictions, the scale of the result is set equal to the contents of the internal

quantity, scale.

Division

The scale of a quotient is the contents of the internal quantity, scale.

Modulo

The scale of a remainder is the sum of the scales of the quotient and the divisor.

Exponentiation

The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.

Square Root

The scale of a square root is set to the maximum of the scale of the argument and the contents of scale.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation is performed, and not rounding.

The contents of scale must be no greater than 99 and no less than 0. It is initially set to 0. If more than 99 fraction digits are needed, it is possible to write a BC program that will handle this.

The internal quantities scale, ibase, and base can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of scale by one, and the line

```
scale
```

causes the current value of scale to be printed.

The value of scale retains its meaning as a number of decimal digits to be retained in internal computation even when ibase or obase are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

BC: A Calculator

9.5 Functions

The name of a function is a single lowercase letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace (}). Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms:

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one **auto** statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each call level are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function, with the single exception that they are given a value on entry to the function. An example of a function definition follows:

```
define a(x,y){
        auto z
        z = x*y
        return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name, followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions may require no arguments, but still perform some useful operation or return a useful result. Such functions are defined and called using parentheses with nothing between them. For example:

```
a ()
```

If the function, "a", is defined as shown above, then the line

```
a(7,3.14)
```

would print the result:

```
21.98
```

Similarly, the line

```
x = a(a(3,4),5)
```

would cause the value of "x" to become 60.

9.6 Subscripted Variables

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable and indicates an array element. The variable name is the name of the array and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted in BC. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, as in:

```
f(a[])
```

Array names may also be declared as automatic in a function definition with the use of empty brackets:

```
define f(a[])
auto a[]
```

When an array name is so used, the entire contents of the array are copied for the use of the function, then thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other context.

9.7 Control Statements

The **if**, **while**, and **for** statements are used to alter the flow within programs or to cause iteration. The range of each of these statements is a following statement or compound statement consisting of a collection of statements enclosed in braces. They are written in the following ways:

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2)statement

if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
expression1rel-opexpression2
```

where the two expressions are related by one of the six relational operators:

```
<  >  <=  >=  ==  !=
```

Note that a double-equals sign (==) stands for "equal to" and an exclamation-equals (!=) stands for "not equal to." The meaning of the remaining relational operators is their normal arithmetic and logical meaning.

Beware of using a single equals sign (=) instead of the double-equals sign (==) in a relational. Both of these symbols are legal, so you will not get a diagnostic message. However, the operation will not perform the intended comparison.

The **if** statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The **while** statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The **for** statement begins by executing expression1. Then the relation is tested and, if true, the statements in the range of the **for** are executed. Then expression2 is executed. The relation is tested, and so on. The typical use of the **for** statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements:

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

Here, the line

```
f(a)
```

will print "a" factorial if "a" is a positive integer. Here is the definition of a function that computes values of the binomial coefficient ("m" and "n" are assumed to be positive integers):

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard to possible truncation errors:

BC: A Calculator

```
scale = 20
define e(x){
        auto a, b, c, d, n
        a = 1
        b = 1
        c = 1
        d = 0
        n = 1
        while(1==1){
                a = a*x
                b = b*n
                c = c + a/b
                n = n + 1
                if(c==d) return(c)
                d = c
        }
}
```

9.8 Language Features

Collected below, are some language features that every user should know about.

Normally, statements are typed one to a line. It is also permissible to type several statements on a line if they are separated by semicolons.

If an assignment statement is parenthesized, it then has a value and can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized:

```
x = a[i=i+1]
```

This causes a value to be assigned to "x" and also increments "i" before it is used as a subscript.

The following constructions work in BC in exactly the same manner as they do in the C language:

```
x=y=z    is the same as  x =(y=z)
x =+ y   is the same as  x = x+y
x =- y   is the same as  x = x-y
x =* y   is the same as  x = x*y
x =/ y   is the same as  x = x/y
x =% y   is the same as  x = x%y
x =^ y   is the same as  x = x^y
x++      is the same as (x=x+1)-1
x--      is the same as (x=x-1)+1
++x      is the same as  x = x+1
--x      is the same as  x = x-1
```

Even if you don't intend to use these constructions, if you type one. inadvertently, something legal but unexpected may happen.

WARNING
In some of these constructions, spaces are significant. There is a real difference between "x=-y" and "x= -y". The first replaces "x" by "x-y" and the second by "-y".

To exit a BC program, type:

```
quit
```

Typing just a "q" will not do.

There is a comment convention identical to that of C. Comments begin with "/*" and end with "*/".

There is a library of math functions that may be obtained by typing

```
bc -l
```

when you invoke BC. This command will load the library functions sine, cosine, arctangent, natural logarithm, exponential, and Bessel functions of integer order. These are named "s", "a", "l", "e", and "j(n,x)", respectively. This library sets scale to 20 by default.

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

9.9 The BC Language

Below is a specification of the BC language.

9.9.1 Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

Comments — Comments are introduced by the characters "/*" and terminated by "*/".

Identifiers — There are three kinds of identifiers: ordinary identifiers, array identifiers and function identifiers. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named "x", an array named "x", and a function named "x", all of which are separate and distinct.

Keywords — The following are reserved keywords:

```
ibase   if
obase   break
scale   define
sqrt    auto
length  return
while   quit
for
```

Constants — Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A-F** are also recognized as digits with values 10-15, respectively.

9.9.2 Expressions

All expressions can be evaluated to a value. The value of an expression is always printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator. There are several types of expressions:

Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

identifiers

Simple identifiers are named expressions. They have an initial value of zero.

array-name[expression]

Array elements are named expressions. They have an initial value of zero.

scale, ibase and obase

The internal registers scale, ibase, and obase are all named expressions. Scale is the number of digits after the decimal point to be retained in arithmetic operations. Scale has an initial value of zero. Ibase and obase are the input and output number radixes respectively. Both ibase and obase have initial values of 10.

Constants

Constants are primitive expressions that evaluate to themselves.

Parenthetic Expressions

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

Function Calls

Function calls are expressions that return values. They are discussed in the following subsection.

9.9.3 Function Calls

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. The syntax is as follows:

function-name ([expression [, expression ...]])

A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement, or is zero if no expression is provided or if there is no return statement.

sqrt(expr) The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of scale whichever is larger.

length(expr) The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale(expr) The result is the scale of the expression. The scale of the result is zero.

9.9.4 Unary Operators

The unary operators bind right to left.

-expr The result is the negative of the expression.

++named-expr The named expression is incremented by one. The result is the value of the named expression after incrementing.

--named-expr The named expression is decremented by one. The result is the value of the named expression after decrementing.

named-expr++ The named expression is incremented by one. The result is the value of the named expression before incrementing.

named-expr-- The named expression is decremented by one. The result is the value of the named expression before decrementing.

9.9.5 Multiplicative Operators

The multiplicative operators (*, /, and %) bind from left to right.

expr*expr The result is the product of the two expressions. If "a" and "b" are the scales of the two expressions, then the scale of the result is:

min(a+b,max(scale,a,b))

expr/expr The result is the quotient of the two expressions. The scale of the result is the value of scale.

expr%expr The modulo operator (%) produces the remainder of the division of the two expressions. More precisely, a%b is a-a/b*b. The scale of the result is the sum of the scale of the divisor and the value of scale.

expr^expr The exponentiation operator binds right to left. The result is the first expression raised to the power of the second expression. The second expression must be an integer. If "a" is the scale of the left expression and "b" is the absolute value of the right expression, then the scale of the result is:

min(a*b,max(scale,a))

9.9.6 Additive Operators

The additive operators bind left to right.

expr+expr The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expr-expr The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

9.9.7 Assignment Operators

The assignment operators listed below assign values to the left-hand named-expression.

named-expr=expr
: This expression results in assigning the value of the expression on the right to the named-expression on the left.

named-expr=+expr
: The result of this expression is equivalent to "named-expr=named-expr+expr."

named-expr=-expr
: The result of this expression is equivalent to "named-expr=named-expr-expr."

named-expr=*expr
: The result of this expression is equivalent to "named-expr=named-expr*expr."

named-expr=/expr
: The result of this expression is equivalent to "named-expr=named-expr/expr."

named-expr=%expr
: The result of this expression is equivalent to "named-expr=named-expr%expr."

named-expr=^expr
: The result of this expression is equivalent to "named-expr=named-expr^expr."

9.9.8 Relations

Unlike all other operators, the relational operators are only valid as the object of an if, while, or inside a for statement. These operators are listed below:

```
expr<expr

expr>expr

expr<=expr

expr>=expr

expr==expr
```

expr!=expr

9.9.9 Storage Classes

There are only two storage classes in BC: global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions.

All identifiers, global and local, have initial values of zero. Identifiers declared as auto are allocated on entry to the function and released on returning from the function. They, therefore, do not retain values between function calls. Auto arrays are specified by the array namer, followed by empty square brackets.

Automatic variables in BC do not work in the same way as in C. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

9.9.10 Statements

Statements must be separated by a semicolon or a new line. Except where altered by control statements, execution is sequential. There are four kinds of statements: expression statements, compound statements, quoted string statements, and built-in statements. Built-in statements include auto, break, **define**, for, if, quit, return, and **while**. Each kind of statement is discussed below:

Expression statements
: When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements
: Statements may be grouped together and used when one statement is expected by surrounding them with curly braces ({ and }).

Quoted string statements
: For example

"string"

prints the string inside the quotes.

Built-in Statements

The syntax for each built-in statement is given below:

auto *identifier* [, *identifier*]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

break

Break causes termination of a **for** or **while** statement.

define ([*parameter* [, *parameter* ...]]){*statements*}

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

for (*expression*; *relation*;*expression*) *statement*

The **for** statement is the same as:

```
first-expression
while(relation) {
        statement
        last-expression
}
```

All three expressions must be present.

if (*relation*) *substatement*

The substatement is executed if the relation is true.

quit

The quit statement stops execution of a BC program and returns control to XENIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

BC: A Calculator

return

return(_expr_**)**

The **return** statement terminates a function, pops its auto variables off the stack, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

while (_relation_**)** _statement_

The statement is executed while the relation is true. The test occurs before each execution of the statement.

APPENDIX A: GLOSSARY

This glossary lists important terms used in this manual. Note that underlined words followed by a number in parentheses refer to entries in the XENIX Reference Manual.

.
: Your current directory has the name "dot" (.) as well as the name printed by the command **pwd**. The current directory is usually the first component of the search path contained in the variable path, thus commands which are in the current directory are found first. The period is also used in separating components of filenames, as in file.txt. The period (.) at the beginning of a component of a pathname is treated specially and not matched by the filename expansion metacharacters (?, *, [, and]). An example of this is the standard file .profile.

..
: Each directory has a file named "dot-dot" (..) that is a reference to its parent directory. After changing into any directory with

```
cd dirname
```

you can return to the parent directory by typing

```
cd ..
```

You can always print the current directory with the **pwd** command.

absolute pathname
: A pathname that begins with a slash (/) is absolute since it specifies the path of directories from the beginning of the entire directory system -- called the root directory. Pathnames which are not absolute are called relative (see the definition of relative pathname).

argument
: Commands in XENIX may accept a list of argument words. Thus, the command

```
echo a b c
```

consists of the command name **echo** and three argument words a, b, and c. The set of

APPENDIX

arguments after the command name is said to be the argument list of the command.

background Commands started without waiting for them to complete are called background commands. You can execute other commands while background commands are running.

base A filename is sometimes thought of as consisting of a base part, before any period (.) character, and an extension, the part after the period. See filename and extension in this glossary for more information.

bin A directory containing executable programs (sometimes called "binaries") and shell scripts is called a bin directory. The standard system bin directories are /bin, containing the most heavily used commands and /usr/bin, which contains most other user programs. Games are kept in the directory /usr/games. You can place programs in any directory. If you wish to execute them often, you should make the name of the directory a component of the variable PATH.

break **Break** is a built-in command used to exit from loops within the control structure of the shell.

built-in A command executed directly by the shell is called a built-in command. Most commands in XENIX are not built into the shell, but exist as files in bin directories. These commands are accessible because the directories in which they reside are named in the PATH variable.

case The shell built-in **case** command allows the shell to select one of a number of sequences of commands based on an argument string.

cat The **cat** program concatenates the contents of a list of files on the standard output. It is usually used to look at the contents of a single file on the terminal.

cd The cd command is used to change your working directory. With no arguments, **cd** changes your working directory to be your HOME directory.

cmp — Cmp is a program that compares files. It is usually used to compare binary files, or to see if two files are identical. For comparing text files the program diff, described in diff(1) is used.

command — A command is an order to execute a given system function. Commands are recognized and executed by the shell. Some commands are part of the shell itself, and therefore, are called built-ins. Other commands are programs or shell procedures residing in a file within the XENIX system.

command name — When a command is issued, it consists of a command name, which is the first word of the command, followed by arguments. The convention in XENIX is that the first word of a command names the function to be performed.

command substitution — Command substitution refers to the replacement of a command enclosed in back quotes (`) with the text output by that command. For example, the following assigns the name of the working directory to the shell variable HERE:

```
HERE=`pwd`
```

component — The parts of pathnames separated by slash characters (/) are called components of that pathname.

control- — Certain special characters, called control characters, are produced by holding down the <CONTROL> key on your terminal and simultaneously pressing another character, much like the <SHIFT> key is used to produce uppercase characters. Thus <CONTROL-C> is produced by holding down the <CONTROL> key while pressing the "c" key.

cp — The cp (copy) program is used to copy the contents of one file into another file. It can also be used to copy several files into a directory.

date — The date command prints the current date and time.

debugging Debugging is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell debugging.

<DEL> The <DEL> key sends an interrupt to the current job and is normally the same as the <INTERRUPT> key. Most interactive commands return to their command level upon receipt of an interrupt, while non-interactive commands usually terminate, returning control to the shell.

detached A command that continues running in the background after you logout is said to be detached.

diagnostic An error message produced by a program is often referred to as a diagnostic. Most error messages are not written to the standard output, since that is often directed away from the terminal. Error messages are instead written to the diagnostic output called "standard error," which is directed to the terminal, but may be redirected if desired.

directory A structure that contains files. When in XENIX, you are always in one particular directory whose name can be printed by the command **pwd**. The **cd** command will change you to another directory. The directory you are in when you first log into XENIX is called your "home" directory.

du The **du** command is a program (described in du(1)) which prints the number of disk blocks is all directories below and including your current working directory.

echo The **echo** command prints its arguments.

else The **else** command is part of the "if-then-else-fi" control command construct.

EOF An end-of-file is generated at the terminal with a <CONTROL-D>, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a pipe receive an end-of-file when the command sending them input completes. Most

commands terminate when they receive an end-of-file.

escape A backslash (\) is used to prevent interpretation of the special meaning of a metacharacter. It is said to "escape" the character from its special meaning. Thus

```
echo \*
```

will echo

```
*
```

while just

```
echo *
```

will echo the names of the files in the current directory. In this example, the backslash (\) escapes the star (*).

There is also a non-printing character called "escape" which we refer to as <ESC> in this manual. This character has no relation to the backslash escape character discussed above.

/etc/passwd This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by colon (:) characters. You can look at this file by typing

```
cat /etc/passwd
```

The commands **finger** and **grep** are often used to search for information in this file. See finger(1), passwd(5), and grep(1) for more details.

exit The **exit** command is used to force termination of a shell script, and is built into the shell.

exit status A command which finishes executing because of some problem can report this fact to the command which invoked it (such as a shell). It does this by returning a non-zero number as its exit status, where a status of zero is considered normal termination. The **exit**

command can be used to force a shell command script to give a non-zero exit status.

expansion The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of expansion. Thus, the replacement of star (*) by a sorted list of files in the current directory is "filename expansion." Expansions are also referred to as substitutions.

expressions Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts.

extension Filenames often consist of a base name and an extension separated by a period (.). By convention, groups of related files often share the same base name. Thus if prog.c is a C program, then the object file for this program would be stored in prog.o. Similarly a paper written with nroff might be stored in paper.n while a troff version of this paper might be kept in paper.t and a list of spelling errors in paper.sp. In these examples, "prog" and "paper" are base names and ".c", ".o", ".n", ".t", and ".sp" are extensions.

filename Each file in XENIX has a name consisting of up to 14 characters not including the slash character (/) which is used in pathname building. Most filenames do not begin with the period character, and contain only letters and digits with perhaps a period separating the base portion of the filename from an extension. Note that XENIX does not interpret the filename characters in any way; base names and extensions are conventions only.

filename expansion
Filename expansion uses the metacharacters (*, ?, \, and, [and]) to provide a convenient mechanism for naming files. Using filename expansion, it is easy to name all the files in the current directory or all files that have a common base name.

flag See switch.

for The for command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list.

foreground When commands execute so that the shell command interpreter waits for them to finish before prompting for another command, these commands are said to be running in the foreground. Foreground jobs can be stopped by signals from the terminal caused by typing different control characters at the keyboard. This is in contrast to commands executing in the background, which the shell does not wait for and which cannot be killed except with the kill command.

grep The **grep** command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file /etc/passwd that contains the string "bill". Actually, **grep** scans for patterns of text called "regular expressions" Thus, the word "grep" stands for "globally find regular expression and print."

head The **head** command prints the first few lines of one or more files. If you have a number of files containing text, only one of which you are interested in, it is sometimes useful to run **head** with the names of these files as arguments. **Head** will usually show enough of what is in these files to let you decide which one you are interested in.

home directory Each user has a home directory, which is given in your entry in the password file, /etc/passwd. This is the directory in which you are placed when you first login. The **cd** command with no arguments takes you back to this directory. The name of this directory is recorded in the shell variable HOME.

if The conditional command, if, is used within shell command scripts to make decisions about what course of action to take next.

input Many commands in XENIX take information from the terminal or from files which they then act on. This information is called input. Commands normally read for input from their standard input which is, by default, the terminal. This standard input can be redirected from a file using the character (<). Many commands will also read from a file specified as an argument. Commands placed in pipelines will read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a filename to use as standard input.

interrupt An interrupt is a signal to a program that is generated by hitting the <INTERRUPT> key. It causes most programs to stop execution. Certain programs, such as the shell and the editors, handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you press <INTERRUPT> because many commands die when they receive an interrupt.

<INTERRUPT> The <INTERRUPT> key causes an interrupt to be sent to the currently running foreground process. This key is normally configured to one of the following keys on a keyboard: <DEL>, <DELETE>, <RUBOUT>, or <CONTROL-C>.

kill The **kill** command sends signals to processes causing them to terminate.

login shell The shell that is started on your terminal when you login is called your "login shell."

logout A login shell will exit when you type <CONTROL-D>, generating an end-of-file.

lpr The **lpr** prints files on the line printer. It is common to use **lpr** as the last component of a pipeline.

ls The ls (list files) command with no argument filenames, prints the names of the files in the current directory. It has a number of useful switches, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories.

mail The **mail** program is used to send and receive messages from other XENIX users.

manual "The manual" is the XENIX Reference Manual. It contains eight sections, the first of which describes each available XENIX program. An on-line version of the manual is accessible through the **man** command, which comes only with the XENIX Text Processing Package.

metacharacter Many characters that are neither letters nor digits have special meaning either to the shell or to XENIX. These characters are called metacharacters. If it is necessary to place these characters in arguments to commands without them having their special meaning, then they must be quoted, or escaped. An example of a metacharacter is the greater-than sign (>), which is used to indicate placement of command output into a file.

mkdir The **mkdir** command is used to create a new directory.

more The program **more** displays the contents of a file on your terminal, allowing you to control how much text is displayed at a time. **More** can move through the file screenful by screenful, line by line, search forward for a string, or start again at the beginning of the file. It is generally the easiest way to view long files.

output Many commands in XENIX result in lines of text which are called their output. This output is usually sent to what is known as the standard output, which is normally connected to the user's terminal. The shell has a syntax using the less-than character (>) for redirecting the standard output of a command to a file. Using the pipe mechanism

and the pipe symbol (|), it is also possible for the standard output of one command to become the standard input of another command. Certain commands such as **lpr** do not send their results to the standard output but rather to more useful places such as the line printer. Similarly, the **write** command sends its output to another user's terminal rather than its standard output. Commands also have a diagnostic output where they write error messages. Normally these go to the terminal even if the standard output has been sent to a file or to another command, but it is possible to redirect error diagnostics along with standard output using a special metanotation.

PATH

The shell has a variable, PATH, which gives the names of the directories in which it searches for commands to execute. The shell always checks first to see if the command it is given is built into the shell. If the command is built in, then then the shell need not search for the command as it can execute the command internally. If the command is not built-in, then the shell searches for a file with the name given in each of the directories in the PATH variable, left to right. Since the normal definition of the PATH variable is

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

the shell normally looks in the current directory, and then in the standard system directories /bin and /usr/bin for the named command. If the command cannot be found the shell prints an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have "execute" permission set. This is normally true because a command of the form

```
chmod +x script
```

was executed to turn this execute permission on.

pathname

A list of names, separated by slash (/) characters, forms a pathname. Each component, between successive slashes, names

a directory in which the next component file resides. Pathnames beginning with a slash are interpreted relative to the root directory in the file system. Other pathnames are interpreted relative to the current directory. The last component of a pathname may name a either a directory or a file.

pipeline A pipeline is a group of commands that are connected together by pipe symbols (|). The standard output of each command is then connected to the standard input of the next.

port The part of a computer system to which each terminal is connected is called a port. Usually the system has a fixed number of ports, some of which may be connected to telephone lines for dial-up access, and some of which may be permanently wired directly to specific terminals.

pr The **pr** command is used to paginate files. At the top of each page is a header giving the name of the file and the date and time at which the file was last modified. Each page also has a page number.

process An instance of a running program is called a process. XENIX assigns each process a unique number from 1-30,000 when it is started -- called the process number. Process numbers can be used to stop individual processes using the kill command.

program Usually synonymous with command; a program is a binary file or shell procedure which performs a system function.

prompt Many programs will print a prompt on the terminal when they expect input. Thus the editor vi(1) will print a colon (:) when it expects input. The shell prompts for input with "$ " and occasionally with "> " when reading commands from the terminal. These prompts are defined by the shell variables named PS1 and PS2, and can be changed by the user.

ps The **ps** command is used to show the processes you are currently running. Each process is

Glossary

shown with its unique process number, the terminal name it is attached to, the state of the process (whether it is running, awaiting some event (sleeping), and whether it is swapped out), and the amount of CPU time it has used so far. The command itself is identified by printing some of the words used when it was invoked.

pwd The **pwd** command prints the full pathname of the current working directory.

quit The quit signal, generated by a <CONTROL-\>, is used to terminate programs which are behaving unreasonably. It normally produces a core image file that is named core in the current directory.

quotation The method of turning off the special meaning of metacharacters either by using the back quote (`) in pairs, or by using the backslash (\) for individual characters.

redirection The routing of input or output from or to a file is known as redirection of input or output.

relative pathname
A pathname which does not begin with a slash (/) is called a relative pathname, since it is interpreted relative to the current, working directory. The first component of such a pathname refers to some file or directory in the working directory, and subsequent components between slashes (/) refer to directories below the working directory. Pathnames that are not relative are called absolute pathnames.

root The directory that is at the top of the entire directory structure is called the root directory, since it is the "root" of the "tree" of directories in the file system. The name used in pathnames to indicate the root is a single slash (/). Pathnames starting with a slash are said to be absolute since they start at the root directory.

script Sequences of shell commands placed in a file are called shell procedures or "scripts." You can perform many tasks using scripts instead

of writing a program in a language such as C.

set The built-in shell command **set** is used to examine the setting of shell variables. For example, typical output might look like this:

```
DEF=/usr/stew/lib/def
HOME=/usr/stew
IFS=

MAIL=/usr/spool/mail/stewk
TERM=h19
PATH=:/bin:/usr/bin:/usr/stew/bin
SHELL=/bin/sh
```

shell A shell is a command language interpreter. It is possible to write and run your own shell, as shells are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular shell called **sh**.

shell script See script.

signal A signal in XENIX is a short message that is sent to a running program to cause some action. Signals are sent either by typing special control characters on the keyboard or by using the **kill** command.

sort The **sort** program sorts a sequence of lines in ways that can be controlled by switches.

special character See metacharacters.

status A command normally returns an exit status when it finishes execution. By convention a status of zero indicates that the command successfully completed execution. Commands may return non-zero status to indicate that some abnormal event has occurred. The status of the last executed command is kept in the shell variable **$?**.

string A sequential group of characters taken together is called a string. Strings can contain any printable characters.

stty The **stty** program changes certain parameters inside XENIX which determine how your

terminal is handled. See stty(1) for a complete description.

substitution — The shell implements a number of substitutions where metacharacters are replaced by other sequences of characters. We also often refer to substitutions as expansions.

switch — Many XENIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as switches, flags, or options. By convention, switches consist of one or more letters preceded by the dash character (-). Thus, the ls (list files) command has an option -s to list the sizes of files. This is specified

```
ls -s
```

termination — When a command finishes executing we say it "terminates." Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate a command by sending it an interrupt or quit signal. The kill program also terminates specified jobs.

then — The then command is part of the shell's "if-then-else-fi" control construct used in shell procedures and scripts.

tty — The word "tty" is an abbreviation for the word "teletype," which is frequently used in XENIX to indicate the port to which a given terminal is connected. The tty command prints the name of the tty or port to which your terminal is presently connected.

XENIX — XENIX is the name of the system of software making up the XENIX system.

variable expansion — See variables and expansion.

variables — Variables have values given by a string of characters. Variables are used to help control the behavior of the shell. See PATH for an example.

wc The **wc** program calculates the number of characters, words, and lines in the files whose names are given as arguments.

while The **while** built-in control construct is used in shell command scripts.

word A sequence of characters that forms an argument to a command is called a word. In general, words are separated by tabs, blanks, or newlines. Any sequence of characters, including spaces, may be made into a word by surrounding it with single quotes ('), except for the quote character itself, which requires special treatment. This process of placing special characters in words without their special meaning is called quoting.

working directory At any given time you are in one particular directory, called your current, or "working directory." This directory's name is printed by the pwd command and the files listed by **ls** are the ones in this directory. You can change working directories using the cd command.

write The write command is used to communicate with other logged in users.

TRS-XENIX System Reference

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities. versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

TRS-XENIX SYSTEM

REFERENCE

CONTENTS

1. Commands

2. System Calls

3. Subroutines

4. Special Files

5. File Formats

6. Games

7. Miscellaney

8. Maintenance

Introduction

This manual is a reference to the commands, facilities, and utilities of the XENIX system. Within the area it surveys, this manual attempts to be complete and concise. This manual does not attempt to provide introductory or tutorial information. See other XENIX manuals for this information.

The volume is divided into eight sections:

Section 1 Commands
Commands are programs intended to be invoked directly by the user. This is in contrast to system calls and subroutines which are intended to be called from within user programs. System startup commands, and commands invoked by only internally by the system are in in Section 8. Some of the entries for this section come only with the XENIX Software Development Package and the XENIX Text Processing Package. Commands generally reside in the directory /bin (for binary programs). Some programs also reside in /usr/bin, to save space in /bin. These directories are searched automatically by the shell command interpreter.

Section 2 System Calls
Section 2 describes the C language interface to each system call. System calls are entries into the XENIX operating system kernel. It is in the kernel that the system's resources are manipulated. Entries for this section come only with the XENIX Software Development Package.

Section 3 Subroutines
An assortment of standard subroutines is available. These subroutines are intended to be included and compiled into C programs. The libraries in which they are kept are described in intro(3). Entries for this section come only with the XENIX Software Development Package.

Section 4 Special Files
This section discusses the characteristics of each system "file" that actually refers to an I/O device.

Section 5 File Formats
This section documents the structure of particular kinds of files. For example, the form

of the output of the loader and assembler is given. Excluded are files used by only one command, for example any program's intermediate files.

Section 6 Games
This section describes the games available in /usr/games.

Section 7 Miscellaneous
This section is mainly a miscellaneous collection of information necessary to writing in various specialized languages. For example character codes, and macro packages for typesetting, are included here.

Section 8 Maintenance
This section discusses procedures not intended for use by the ordinary user. These procedures often involve use of commands of section 1, where an attempt has been made to single out peculiarly maintenance-flavored commands by marking them 1M.

Page Format

Each section consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, together with the section number, and sometimes a letter characteristic of a subcategory, e.g. graphics is 1G, and the math library is 3M. Entries within each section are alphabetized. The page numbers of each entry start at 1.

All entries are based on a common format, not all of whose subsections will always appear.

The **NAME** subsection lists the exact names of the commands and subroutines covered under the entry and gives a very short description of their purpose.

The SYNTAX summarizes usage of the program being described.

The **DESCRIPTION** subsection discusses in detail the subject at hand.

The **FILES** subsection gives the names of files that are built into the program or that are closely related to the manual entry.

A **SEE ALSO** subsection gives pointers to related information.

A **DIAGNOSTICS** subsection discusses the error messages and other diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.

The **NOTES** subsection gives known bugs, limitations, and other programmer notes.

At the end of this volume is a permuted index derived from the table of contents. Within each index entry, the title of the writeup to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands which exist only to exercise a particular system call.

Notational Conventions

The following conventions are used throughout this manual:

Boldface words are considered literals, and are typed just as they appear.

Square brackets [ ] around an argument indicate that the argument is optional. When an argument is given as `name', it always refers to a file name.

Ellipses `...' are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. A dash (-) is often taken to mean some sort of option-specifying argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with a dash.

NAME

intro - introduction to commands

SYNTAX

This section gives the syntax for the command

DESCRIPTION

This section describes publicly accessible commands in alphabetic order. Certain distinctions of purpose are made in the headings:

(1) Commands of general utility.

(1S) Commands for software development. These may not be available if you own only the basic XENIX package.

(1T) Commands for text processing and formatting. These may not be available if you own only the basic XENIX package.

(1M) Commands used primarily for system maintenance. These are available on all systems.

The word 'XENIX-nn' at the foot of a page means that the command is either unique or different for this version of XENIX.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of 'normal' termination) one supplied by the program, see wait and exit(2). The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously 'exit code', 'exit status' or 'return code', and is described only where special conventions are involved.

NAME

ac - login accounting

SYNTAX

ac [**-w** wtmp] [**-p**] [**-d**] [people] ...

DESCRIPTION

Ac produces a printout giving connect time for each user who has logged in during the life of the current wtmp file. A total is also produced. **-w** is used to specify an alternate wtmp file. **-p** prints individual totals; without this option, only totals are printed. **-d** causes a printout for each midnight to midnight period. Any people will limit the printout to only the specified login names. If no wtmp file is given, /usr/adm/wtmp is used.

The accounting file /usr/adm/wtmp is maintained by init and login. Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

FILES

/usr/adm/wtmp

SEE ALSO

init(8), login(1), utmp(5).

NAME

accton - turn on system accounting

SYNTAX

/etc/accton [file]

DESCRIPTION

With an argument naming an existing file, accton causes system accounting information for every process executed to be placed at the end of the file. If no argument is given, accounting is turned off.

Sa(1) performs actual accounting funtions.

FILES

/usr/adm/acct	raw accounting
/usr/adm/savacct	summary
/usr/adm/usracct	per-user summary

SEE ALSO

ac(1), acct(2), sa(1)

NAME

adb - debugger

SYNTAX

adb [-w] [objfil [corfil]]

DESCRIPTION

Adb is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of XENIX programs.

Objfil is normally an executable program file, preferably containing a symbol table; if not then the symbolic features of adb cannot be used although the file can still be examined. The default for objfil is **a.out**. Corfil is assumed to be a core image file produced after executing objfil; the default for corfil is **core**.

Requests to adb are read from the standard input and responses are to the standard output. If the **-w** flag is present then both objfil and corfil are created if necessary and opened for reading and writing so that files can be modified using adb. Adb ignores QUIT; INTERRUPT causes return to the next adb command.

In general requests to adb are of the form

[address] [, count] [command] [;]

If address is present then dot is set to address. Initially dot is set to 0. For most commands count specifies how many times the command will be executed. The default count is 1. Address and count are expressions.

The interpretation of an address depends on the context it is used in. If a subprocess is being debugged then addresses are interpreted in the usual way in the address space of the subprocess. For further details of address mapping see ADDRESSES.

EXPRESSIONS

. The value of dot.

\+ The value of dot incremented by the current increment.

^ The value of dot decremented by the current increment.

" The last address typed.

integer

An octal number if integer begins with a 0; a decimal number, if preceded by 0t or 0d; otherwise a hexidecimal number. Hexidecimal may also be input with a preceding number sign '#' or 0x. #; otherwise a decimal number.

integer.fraction
A 32 bit floating point number.

'cccc' The ASCII value of up to 4 characters. \ may be used to escape a '.

< name The value of name, which is either a variable name or a register name. Adb maintains a number of variables (see VARIABLES) named by single letters or digits. If name is a register name then the value of the register is obtained from the system header in corfil.

symbol A symbol is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The value of the symbol is taken from the symbol table in objfil. An initial _ or ~ will be prepended to symbol if needed.

symbol
In C, the 'true name' of an external symbol begins with an underscore (_). It may be necessary to use this name to disinguish it from the internal or hidden variables of a program.

(exp) The value of the expression exp.

Monadic operators

*exp The contents of the location addressed by exp in corfil.

@exp The contents of the location addressed by exp in objfil.

-exp Integer negation.

~exp Bitwise complement.

Dyadic operators are left associative and are less binding than monadic operators.

e1+e2 Integer addition.

e1-e2 Integer subtraction.

e1*e2 Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

e1#e2 E1 rounded up to the next multiple of e2.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands '?' and '/' may be followed by '*'; see ADDRESSES for further details.)

?f Locations starting at address in objfil are printed according to the format f.

/f Locations starting at address in corfil are printed according to the format f.

=f The value of address itself is printed in the styles indicated by the format f. (For i format '?' is printed for the parts of the instruction that reference subsequent words.)

A format consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format dot is incremented temporarily by the amount given for each format letter. If no format is given then the last format is used. The format letters available are as follows.

o	2	Print 2 bytes in octal. All octal numbers output by adb are preceded by 0.
O	4	Print 4 bytes in octal.
q	2	Print in signed octal.
Q	4	Print long signed octal.
d	2	Print in decimal.
D	4	Print long decimal.
x	2	Print 2 bytes in hexadecimal.
X	4	Print 4 bytes in hexadecimal.
u	2	Print as an unsigned decimal number.
U	4	Print long unsigned decimal.
f	4	Print the 32 bit value as a floating point number.
F	8	Print double floating point.
b	1	Print the addressed byte in octal.
c	1	Print the addressed character.
C	1	Print the addressed character using the following escape convention. Character values 000 to 040

are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.

s n Print the addressed characters until a zero character is reached.

S n Print a string using the @ escape convention. n is the length of the string including its zero terminator.

Y 4 Print 4 bytes in date format (see ctime(3)).

i n Print as machine instructions. n is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.

a 0 Print the value of dot in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

p 2 Print the addressed value in symbolic form using the same rules for symbol lookup as a.

t 0 When preceded by an integer tabs to the next appropriate tab stop. For example, 8t moves to the next 8-space tab stop.

r 0 Print a space.

n 0 Print a newline.

"..." 0
Print the enclosed string.

^ Dot is decremented by the current increment. Nothing is printed.

\+ Dot is incremented by 1. Nothing is printed.

\- Dot is decremented by 1. Nothing is printed.

newline
If the previous command temporarily incremented dot, make the increment permanent. Repeat the previous command with a count of 1.

[?/]l value mask
Words starting at dot are masked with mask and compared with value until a match is found. If L is used then the match is for 4 bytes at a time instead of 2. If no match is found then dot is unchanged; otherwise dot is set to the matched location. If mask is omitted then -1 is used.

[?/]w value ...
Write the 2-byte value into the addressed location. If the command is W, write 4 bytes. Odd addresses are not

allowed when writing to the subprocess address space.

[?/]m b1 e1 f1[?/]
New values for (b1, e1, f1) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second segment (b2,e2,f2) of the mapping is changed. If the list is terminated by '?' or '/' then the file (objfil or corfil respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to objfil.)

>name
Dot is assigned to the variable or register named.

! A shell is called to read the rest of the line following '!'.

$modifier
Miscellaneous commands. The available modifiers are:

- <f Read commands from the file f and return.
- >f Send output to the file f, which is created if it does not exist.
- r Print the general registers and the instruction addressed by pc. Dot is set to pc.
- b Print all breakpoints and their associated counts and commands.
- c C stack backtrace. If address is given then it is taken as the address of the current frame.
- e The names and values of external variables are printed.
- w Set the page width for output to address (default 80).
- s Set the limit for symbol matches to address (default 255).
- o All integers input are regarded as octal.
- d Reset integer input as described in EXPRESSIONS.
- q Exit from adb.
- v Print all non zero variables in octal.
- m Print the address map.

:modifier
Manage a subprocess. Available modifiers are:

- bc Set breakpoint at address. The breakpoint is executed count-1 times before causing a stop. Each time the breakpoint is encountered the command c is executed. If this command sets dot to zero then the breakpoint causes a stop.
- d Delete breakpoint at address.

r Run objfil as a subprocess. If address is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. count specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

cs The subprocess is continued with signal s c s, see signal(2). If address is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for r.

ss As for **c** except that the subprocess is single stepped count times. If there is no current subprocess then objfil is run as a subprocess as for r. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

k The current subprocess, if any, is terminated.

VARIABLES

Adb provides a number of variables. Named variables are set initially by adb but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

On entry the following are set from the system header in the corfil. If corfil does not appear to be a **core** file then these values are set from objfil.

b The base address of the data segment.
d The data segment size.
e The entry point.
s The stack segment size.
t The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (b1, e1, f1) and (b2, e2, f2) and the file address corresponding to a written address is calculated as follows.

b1<address<e1 => file address=address+f1-b1, otherwise,

b2<address<e2 => file address=address+f2-b2,

otherwise, the requested address is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a ? or / is followed by an * then only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, b1 is set to 0, e1 is set to the maximum file size and f1 is set to 0; in this way the whole file can be examined with no address translation.

So that adb may be used on large files all appropriate values are kept as signed 32 bit integers.

FILES

/dev/mem
/dev/swap
a.out
core

SEE ALSO

ptrace(2), a.out(5), core(5)

DIAGNOSTICS

'Adb' when there is no current command or format. Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero status.

NOTES

A breakpoint set at the entry point is not effective on initial entry to the program.
When single stepping, system calls do not count as an executed instruction.
Local variables whose names are the same as an external variable may foul up the accessing of the external.

NAME

ar - archive and library maintainer

SYNTAX

ar key [posname] afile name ...

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose.

Key is one character from the set drqtpmx, optionally concatenated with one or more of vuaibcl. Afile is the archive file. The names are constituent files in the archive file. The meanings of the key characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character u is used with r, then only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set abi is used, then the posname argument must be present and specifies that new files are to be placed after (a) or before (b or i) posname. Otherwise new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the posname argument must be present and, as in r, specifies where the files are to be moved.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does x alter the archive file.

v Verbose. Under the verbose option, ar gives a file-by-file description of the making of a new archive file

from the old archive and the constituent files. When used with t, it gives a long listing of all information about the files. When used with p, it precedes each file with a name.

c Create. Normally ar will create afile when it needs to. The create option suppresses the normal message that is produced when afile is created.

l Local. Normally ar places its temporary files in the directory /tmp. This option causes them to be placed in the local directory.

FILES

/tmp/v* temporaries

SEE ALSO

ld(1S), ar(5S), lorder(1S)

NOTES

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

arcv - convert format of archives

SYNTAX

arcv file ...

DESCRIPTION

Arcv converts archive files (see ar(1S), ar(5)) from PDP-11 format to M68000 format. The conversion is done in place, and the command refuses to alter a file not in PDP-11 archive format.

PDP-11 archives are indicated with a magic number whose bytes have been swapped; M68000 archives have 0177545 as the magic number.

FILES

/tmp/v*, temporary copy

SEE ALSO

ar(1S), ar(5)

NAME

as - assembler

SYNTAX

as [**-l**] [**-o** objfile] [**-g**] file.s

DESCRIPTION

As assembles the named file. If the optional first argument **-l** is used, an assembly listing is produced and written to file.L. This includes the source, the assembled code, and any assembly errors.

The output of the assembly is left on the file objfile; if that is omitted, file.o is used. If the optional **-g** flag is given, undefined symbols will be treated as externals. Arguments may appear in any order, except that **-o** must immediatly precede objfile, Flag arguments may be bunched.

Objfile is made executable if no errors occurred during the assembly, and if there are no undefined symbols.

If file does not end with a **.s** then a warning is produced, but the assembly is run normally anyway. The objfile will have the last character replaced by **o** and the listing file will have the last character replaced by **L**

FILES

/tmp/A68tmpr* temporary

SEE ALSO

ld(1S), nm(1S), adb(1S), a.out(5)

NAME

asktime - prompt for the correct time of day

SYNTAX

/etc/asktime

DESCRIPTION

This command prompts for the time of day. You must enter a legal time according to the proper format as defined below

```
[yymmdd]hhmm[.ss]
```

where yy is a two digit year, mm is a two digit month, dd is a two digit day of the month, hh is the hour in military time, mm is the minutes, and ss is seconds.

EXAMPLES

This example sets the new time, date, and year to "9:23.45 January 1, 1983".

```
I think it's Wed Nov  3 14:36:23 PST 1982
Enter time ([yymmdd]hhmm[.ss]): 8301010923.45
```

This example simply resets the time of day to "9:23" and leaves the date and year as they are:

```
I think it's Wed Nov  3 14:36:23 PST 1982
Enter time ([yymmdd]hhmm[.ss]): 0923
```

DIAGNOSTICS

If you enter an illegal time, asktime prompts with:

```
Try again:
```

NOTES

Asktime is normally performed immediately after the system is booted; however, it may be executed at any time. The command is privileged, and can only be executed by the superuser.

NAME

at - execute commands at a later time

SYNTAX

at time [day] [file]

DESCRIPTION

At makes a copy of the named file to be used as input to the shell at a specified later time. If no named file is given, then file is assumed to be the standard input by default (i.e., your terminal). As the first line of the copied file, at inserts a cd(1) command to the current directory. This is followed by assignments to all environment variables. When the script is run, it uses the user and group ID of the creator of the copy file.

The time is 1 to 4 digits, with an optional following 'A', 'P', 'N' or 'M' for AM, PM, noon or midnight. One and two digit numbers are taken to be hours, three and four digits to be hours and minutes. If no letters follow the digits, a 24 hour clock time is understood.

The optional day is either (1) a month name followed by a day number, or (2) a day of the week; if the word 'week' follows invocation is moved seven days further off. Names of months and days may be recognizably truncated. Examples of legitimate commands are

```
at 8am jan 24
at 1530 fr week
```

At programs are executed by periodic execution of the command /usr/lib/atrun from cron(8). The granularity of at depends upon how often atrun is executed.

Standard output or error output is lost unless redirected.

FILES

/usr/spool/at/yy.ddd.hhhh.uu
activity to be performed at hour hhhh of year day ddd of year yy. uu is a unique number.
/usr/spool/at/lasttimedone contains hhhh for last hour of activity.
/usr/spool/at/past directory of activities now in progress
/usr/lib/atrun program that executes activities that are due
pwd(1)

SEE ALSO

calendar(1), cron(8)

DIAGNOSTICS

Complains about various syntax errors and times out of

range.

NOTES

Due to the granularity of the execution of /usr/lib/atrun, there may be problems in scheduling things almost exactly 24 hours into the future.

NAME

awk - pattern scanning and processing language

SYNTAX

awk [**-F**c] [prog] [file] ...

DESCRIPTION

Awk scans each input file for lines that match any of a set of patterns specified in prog. With each pattern in prog there can be an associated action that will be performed when a line of a file matches the pattern. The set of patterns may appear literally as prog, or in a file specified as -f file.

Files are read in order; if there are no files, the standard input is read. The file name '-' means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, vide infra.) The fields are denoted $1, $2, ... ; $0 refers to the entire line.

A pattern-action statement has the form

```
pattern { action }
```

A missing { action } means print the line; a missing pattern always matches.

An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next # skip remaining patterns on this input line
exit # skip the rest of the input
```

Statements are terminated by semicolons, newlines or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, -, *, /, %, and concatenation (indicated by a blank). The C operators ++, --,

+=, -=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted "...".

The print statement prints its arguments on the standard output (or on a file if >file is present), separated by the current output field separator, and terminated by the output record separator. The printf statement formats its expression list according to the format (see printf(3)).

The built-in function length returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions exp, log, sqrt, and int. The last truncates its argument to an integer. substr(s, m, n) returns the n-character substring of s that begins at position m. The function sprintf(fmt, expr, expr, ...) formats the expressions according to the printf(3) format given by fmt and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in egrep. Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

where a relop is any of the six relational operators in C, and a matchop is either ~ (for contains) or !~ (for does not contain). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character c may be used to separate the fields by starting the program with

```
BEGIN { FS = "c" }
```

or by using the -Fc option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default "%.6g").

EXAMPLES

The following examples all assume that input is read from a file. If, instead, input is read from the shell command line, then special characters need to be escaped or quoted.

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

```
      { s += $1 }
END   { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

SEE ALSO

lex(1), sed(1)
A. V. Aho, B. W. Kernighan, P. J. Weinberger, Awk - a pattern scanning and processing language

NOTES

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate "" to it.

NAME

basename - strip filename affixes

SYNTAX

basename string [suffix]

DESCRIPTION

Basename deletes any prefix ending in '/' and the suffix, if present in string, from string, and prints the result on the standard output. It is normally used inside substitution marks ` ` in shell procedures.

This shell procedure invoked with the argument /usr/lib/book.n formats the named file and moves the output to book.txt in the current directory:

```
nroff $1 >`basename $1 .n`.txt
```

This is a common way of handling default filename extensions.

SEE ALSO

sh(1)

NAME

bc - arbitrary-precision arithmetic language

SYNTAX

bc [-c] [-l] [file ...]

DESCRIPTION

Bc is an interactive processor for a language which resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The -l argument stands for the name of an arbitrary precision math library. The syntax for bc programs is as follows; L means letter a-z, E means expression, S means statement.

Comments
are enclosed in /* and */.

Names
simple variables: L
array elements: L [E]
The words 'ibase', 'obase', and 'scale'

Other operands
arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
length (E) number of significant decimal digits
scale (E) number of digits right of decimal point
L (E , ... , E)

Operators
+ - * / % ^ (% is remainder; ^ is power)
++ -- (prefix and postfix; apply to names)
== <= >= != < >
= =+ =- =* =/ =% =^

Statements
E
{ S ; ... ; S }
if (E) S
while (E) S
for (E ; E ; E) S
null statement
break
quit

Function definitions
define L (L ,..., L) {
auto L, ... , L
S; ... S

```
        return ( E )
    }
```

Functions in -l math library

```
    s(x) sine
    c(x) cosine
    e(x) exponential
    l(x) log
    a(x) arctangent
    j(n,x)    Bessel function
```

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or newlines may separate statements. Assignment to scale influences the number of digits to be retained on arithmetic operations in the manner of dc(1). Assignments to ibase or obase set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. 'Auto' variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

For example

```
scale = 20
define e(x){
    auto a, b, c, i, s
    a = 1
    b = 1
    s = 1
    for(i=1; 1==1; i++){
        a = a*x
        b = b*i
        c = a/b
        if(c == 0) return(s)
        s = s+c
    }
}
```

defines a function to compute an approximate value of the exponential function and

```
    for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

Bc is actually a preprocessor for dc(1), which it invokes automatically, unless the -c (compile only) option is present. In this case the dc input is sent to the standard output instead.

FILES

/usr/lib/lib.b mathematical library
dc(1) desk calculator proper

SEE ALSO

dc(1)
L. L. Cherry and R. Morris, BC - An arbitrary precision desk-calculator language

NOTES

No &&, ||, or ! operators.
For statement must have all three E's.
Quit is interpreted when read, not when executed.

NAME

cal - print calendar

SYNTAX

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. Year can be between 1 and 9999. The month is a number between 1 and 12.

NOTES

The year is always considered to start in January even though this is historically inaccurate.
Beware that 'cal 83' refers to the year 83 and not to 1983.

NAME

calendar - reminder service

SYNTAX

calendar [-]

DESCRIPTION

Calendar consults the file 'calendar' in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates such as 'Dec. 7,' 'december 7,' '12/7,' etc., are recognized, but not '7 December' or '7/12'. On weekends 'tomorrow' extends through Monday.

When an argument is present, calendar does its job for every user who has a file 'calendar' in his login directory and sends him any positive results by mail(1). Normally this is done daily in the wee hours under control of cron(8).

FILES

calendar
/usr/lib/calendar to figure out today's and tomorrow's dates
/etc/passwd
/tmp/cal*
egrep, sed, mail subprocesses

SEE ALSO

at(1), cron(8), mail(1)

NOTES

Calendar's extended idea of 'tomorrow' doesn't account for holidays.

NAME

cat - catenate and print

SYNTAX

cat [-u] file ...

DESCRIPTION

Cat reads each file in sequence and writes it on the standard output. Thus

```
cat file
```

prints the file and

```
cat filel file2 >file3
```

concatenates the first two files and places the result on the third.

If no file is given, or if the argument '-' is encountered, cat reads from the standard input. Output is buffered in 512-byte blocks unless the -u option is present.

SEE ALSO

more(1), pr(1), cp(1)

WARNINGS

Beware of 'cat a b >a' and 'cat a b >b', which destroy input files before reading them. Also 'cat * >output' will cause output to grow infinitely.

NAME

cb - C program beautifier

SYNTAX

cb

DESCRIPTION

Cb places a copy of the C program from the standard input on the standard output with spacing and indentation that displays the structure of the program.

NOTES

Output is not always as one would desire.

NAME

cc - C compiler

SYNTAX

cc [option] ... file ...

DESCRIPTION

Cc is the XENIX M68000 C compiler. Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with '.s' are taken to be assembly source programs and are assembled, producing a '.o' file.

The following options are interpreted by cc. See ld(1S) for load-time options.

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-O Invoke an object-code optimizer.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

-o output
Name the final output file output. If this option is used the file 'a.out' will be left undisturbed.

-Dname=def
-Dname Define the name to the preprocessor, as if by '#define'. If no definition is given, the name is defined as 1.

-Uname Remove any initial definition of name.

-Idir '#include' files whose names do not begin with '/' are always sought first in the directory of the file argument, then in directories named in **-I** options, then in directories on a standard list.

-t1 replace the compiler phase with a program called **c68** from the current directory.

-t2 replace the object code optimizer phase with a program called **c68o** from the current directory.

-K Do not generate stack probes. Stack probes are necessary for XENIX user programs to assure proper stack growth.

-w Suppress compiler warning messages

Other arguments
are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier cc run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name a.out.

FILES

file.c	input file
file.o	object file
a.out	loaded output
file.[isx]	temporaries for cc
/lib/cpp	preprocessor
/lib/c68	compiler for cc
/lib/c68o	optional optimizer
/lib/crt0.o	runtime startoff
/lib/libc.a	standard library, see intro(3)
/usr/include	standard directory for '#include' files

SEE ALSO

B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, 1978
D. M. Ritchie, C Reference Manual
adb(1S), ld(1S)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler, loader Of these, the most mystifying are from the assembler, as(1), which produces line number reports based on the generated code, which is only loosely related to the source line number. running the compiler with the -S option and assembling the result by hand may help you resolve the difficulty.

NOTES

The -f (software floating point) and **-p** (profiling) options are not currently implemented.

NAME

cd, chdir - change working directory

SYNTAX

cd [directory]
chdir [directory]

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in directory.

Because a new process is created to execute each command, cd would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell. Note that chdir is a synonym for cd.

SEE ALSO

sh(1), pwd(1), chdir(2)

NAME

checkeq - check eqn usage

SYNTAX

checkeq [file] ...

DESCRIPTION

The program checkeq reports missing or unbalanced delimiters and .EQ/.EN pairs. It is useful to quickly check usage before processing files using eqn, because of the time saved.

SEE ALSO

eqn(1T), neqn(1T), troff(1T), tbl(1T), ms(7), eqnchar(7)

NAME

chgrp - change group

SYNTAX

chgrp group file ...

DESCRIPTION

Chgrp changes the group-ID of the files to group. The group may be either a decimal GID or a group name found in the group-ID file.

To simplify accounting procedures, only the super-user can change owner or group.

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(1), chown(2), passwd(5), group(5)

NAME

chmod - change mode

SYNTAX

chmod mode file ...

DESCRIPTION

The mode of each named file is changed according to mode, which may be absolute or symbolic. An absolute mode is an octal number constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit, see chmod(2)
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic mode has the form:

[who] op permission [op permission] ...

The who part is a combination of the letters u (for user's permissions), g (group) and o (other). The letter a stands for ugo. If who is omitted, the default is a but the setting of the file creation mask (see umask(2)) is taken into account.

Op can be + to add permission to the file's mode, - to take away permission and = to assign permission absolutely (all other bits will be reset).

Permission is any combination of the letters r (read), w (write), x (execute), s (set owner or group id) and t (save text - sticky). Letters u, g or o indicate that permission is to be taken from the current mode. Omitting permission is only useful with = to take away all permissions.

The first example denies write permission to others, the second makes a file executable:

```
chmod o-w file
chmod +x file
```

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter s is only useful with u or g.

Only the owner of a file (or the super-user) may change its mode.

SEE ALSO

ls(1), chmod(2), chown (1), stat(2), umask(2)

NAME

chown - change owner

SYNTAX

chown owner file ...

DESCRIPTION

Chown changes the owner of the files to owner. The owner may be either a decimal UID or a login name found in the password file.

To simplify accounting procedures, only the super-user can change owner or group.

FILES

/etc/passwd
/etc/group

SEE ALSO

chgrp(1), chown(2), passwd(5), group(5)

NAME

clri - clear i-node

SYNTAX

clri filesystem i-number ...

DESCRIPTION

Clri writes zeros on the i-nodes with the decimal i-numbers on the filesystem. After clri, any blocks in the affected file will show up as 'missing' in an icheck(1) of the filesystem.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

SEE ALSO

icheck(1)

NOTES

If the file is open, clri is likely to be ineffective.

NAME

cmp - compare two files

SYNTAX

cmp [**-l**] [**-s**] filel file2

DESCRIPTION

The two files are compared. (If filel is '-', the standard input is used.) Under default options, cmp makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

Options:

-l Print the byte number (decimal) and the differing bytes (octal) for each difference.

-s Print nothing for differing files; return codes only.

SEE ALSO

diff(1), comm(1)

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

col - filter reverse line feeds

SYNTAX

col [-bfx]

DESCRIPTION

Col reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ESC-7 in ASCII) and by forward and reverse half line feeds (ESC-9 and ESC-8). Col is particularly useful for filtering multicolumn output made with the '.rt' command of nroff(1T) and output resulting from use of the tbl(1T) preprocessor.

Although col accepts half line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full line boundary. This treatment can be suppressed by the -f (fine) option; in this case the output from col may contain forward half line feeds (ESC-9), but will still never contain either kind of reverse line motion.

If the -b option is given, col assumes that the output device in use is not capable of backspacing. In this case, if several characters are to appear in the same place, only the last one read will be taken.

The control characters SO (ASCII code 017), and SI (016) are assumed to start and end text in an alternate character set. The character set (primary or alternate) associated with each printing character read is remembered; on output, SO and SI characters are generated where necessary to maintain the correct treatment of each character.

Col normally converts white space to tabs to shorten printing time. If the -x option is given, this conversion is suppressed.

All control characters are removed from the input except space, backspace, tab, return, newline, ESC (033) followed by one of SI, SO, and VT (013). This last character is an alternate form of full reverse line feed, for compatibility with some other hardware conventions. All other nonprinting characters are ignored.

SEE ALSO

tbl(1T), troff(1T)

NOTES

Can't back up more than 128 lines.
No more than 800 characters, including backspaces, on a line.

NAME

comm - select or reject lines common to two sorted files

SYNTAX

comm [- [123]] file1 file2

DESCRIPTION

Comm reads file1 and file2, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in file1; lines only in file2; and lines in both files. The filename '-' means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm -12** prints only the lines common to the two files; **comm -23** prints only lines in the first file but not in the second; **comm -123** is a no-op.

SEE ALSO

cmp(1), diff(1), uniq(1)

NAME

config - configure a XENIX system

SYNTAX

/etc/config [-t] [-c file] [-m file] dfile

DESCRIPTION

Config is a program that takes a description of a XENIX system and generates a file which is a C program defining the configuration tables for the various devices on the system.

The -c option specifies the name of the configuration table file; c.c is the default name.

The -m option specifies the name of the file that contains all the information regarding supported devices; /etc/master is the default name. This file is supplied with the XENIX system and should not be modified unless the user fully understands its construction.

The -t option requests a short table of major device numbers for character and block type devices. This can facilitate the creation of special files.

The user must supply dfile; it must contain device information for the user's system. This file is divided into two parts. The first part contains physical device specifications. The second part contains system-dependent information. Any line with an asterisk (*) in column 1 is a comment.

All configurations are assumed to have a set of required devices which must be present to run XENIX such as the system clock. These devices must not be specified in dfile.

First Part of dfile

Each line contains two fields, delimited by blanks and/or tabs in the following format:

devname number

where devname is the name of the device (as it appears in the /etc/master device table), and number is the number (decimal) of devices associated with the corresponding controller; number is optional, and if omitted, a default value which is the maximum value for that controller is used.

There are certain drivers that may be provided with the system, that are actually pseudo-device drivers; that is, there is no real hardware associated with the driver. Drivers of this type are identified on their respective manual entries.

Second Part of dfile
The second part contains three different types of lines. Note that all specifications of this part are required, although their order is arbitrary.

1. Root/pipe device specification

Each line has three fields:

```
root devname   minor
pipe devname   minor
```

where minor is the minor device number (in octal).

2. Swap device specification

One line that contains five fields as follows:

```
swap devname   minor     swplo     nswap
```

where swplo is the lowest disk block (decimal) in the swap area and nswap is the number of disk blocks (decimal) in the swap area.

3. Parameter specification

A number of lines of two fields each as follows (number is decimal):

```
buffers        number
inodes         number
files          number
mounts         number
swapmap        number
pages          number
calls          number
procs          number
maxproc        number
texts          number
clists         number
locks          number
timezone       number
daylight       0 or 1
```

Suppose we wish to configure a system with the following devices:

one S4 disk drive controller with 1 drive
one F8 floppy disk drive controller with 1 driver

We must also specify the following parameter information:

root device is an S4 (pseudo disk 3)
pipe device is an S4 (pseudo disk 3)
swap device is an S4 (pseudo disk 2)
with a swplo of 1 and an nswap of 2300

```
number of buffers is 50
number of processes is 50
maximum number of processes per user ID is 15
number of mounts is 8
number of inodes is 120
number of files is 120
number of calls is 30
number of texts is 35
number of character buffers is 150
number of swapmap entries is 50
number of memory pages is 512
number of file locks is 100
timezone is pacific time
daylight time is in effect
```

The actual system configuration would be specified as follows:

```
s4    1
f8    1
root    s4    3
pipe    s4    3
swap    s4    2      0    2300
* Comments may be inserted in this manner
buffers 50
procs       150
maxproc 15
mounts      8
inodes      120
files       120
calls       30
texts       35
clists      150
swapmap     50
pages (1024/2);
locks 100
timezone (8*60)
daylight 1
```

FILES

/etc/master	default input master device table
c.c	default output configuration table file

SEE ALSO

master(5).

DIAGNOSTICS

Diagnostics are routed to the standard output and are self-explanatory.

NOTES

The -t option does not know about devices that have aliases. For example, an RP06 (an alias for an RP04) will show up as

an RP04; however, the major device numbers are always correct.

NAME

copy - copy groups of files

SYNTAX

copy [option] ... source ... dest

DESCRIPTION

The copy command copies the contents of directories to another directory. It is possible to copy whole file systems since directories are made when needed.

If files, directories, or special files do not exist at the destination, then they are created with the same modes and flags of the source. In addition, the super-user may set the user and group ids. The owner and mode will not be changed if the destination file exists. Note that there may be more than one source directory. If so, then the effect is the same as if the copy command had been issued, each with only one source.

All of the options must be given as separate arguments and they may appear in any order even after the other arguments. The arguments are:

-a Asks the user before attempting a copy. If the response does not begin with a 'y', then a copy will not be done. This option also sets the '-ad' flag.

-l Uses links instead whenever they can be used. Otherwise a copy is done. Note that links are never done for special files or directories.

-n Requires the destination file to be new. If not, then the copy command will not change the destination file. Of course the '-n' flag is meaningless for directories. For special files a '-n' flag is assumed (i.e., the destination of a special file must not exist).

-o Only the super user may set this option. If set then every file copied will have its owner and group set to those of the source. If not set, then the owner will be that of the user who invoked the program.

-m If set then every file copied will have its modification time and access time set to that of the source. If not set, then the modification time will be set to the time of the copy.

-r If set, then every directory is recursively examined as it is encountered. If not set then any

directories that are found will be ignored.

-ad Asks the user whether a '-r' flag applies when a directory is discovered. If the answer does not begin with a 'y', then the directory will be ignored.

-v If the verbose option is set, then all kinds of messages will be printed that reveal what the program is doing.

source This may be a file, directory or special file. It must exist. If it is not a directory, then the results of the command will be the same as for the cp command.

dest The destination must be either a file or directory different from the source.

If the source and destination are anything but directories, then copy will act just like a cp command. If both are directories, then copy will copy each file into the destination directory according to the flags that have been set.

DIAGNOSTICS

Should be self-explanatory

NAME

cp - copy

SYNTAX

cp file1 file2

cp file ... directory

DESCRIPTION

File1 is copied onto file2. The mode and owner of file2 are preserved if it already existed; the mode of the source file is used otherwise.

In the second form, one or more files are copied into the directory with their original file-names.

Cp refuses to copy a file onto itself.

SEE ALSO

cat(1), pr(1), mv(1), copy(1)

NAME

crypt - encode/decode

SYNTAX

crypt [password]

DESCRIPTION

Crypt reads from the standard input and writes on the standard output. The password is a key that selects a particular transformation. If no password is given, crypt demands a key from the terminal and turns off printing while the key is being typed in. Crypt encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

will print the clear.

Files encrypted by crypt are compatible with those treated by the editor ed in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; 'sneak paths' by which keys or cleartext can become visible must be minimized.

Crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e. to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lower-case letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the crypt command, it is potentially visible to users executing ps(1) or a derivative. To minimize this possibility, crypt takes care to destroy any record of the key immediately upon entry. No doubt the choice of keys and key security are the most vulnerable aspect of crypt.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(8)

NOTES

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, Bell Telephone Laboratories and the Microsoft Corporation assume no responsibility for their use by the recipient. Further, neither Bell Laboratories nor the Microsoft Corporation assumes any obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

NAME

csh - a shell (command interpreter) with C-like syntax

SYNTAX

csh [**-cefinstvVxX**] [arg ...]

DESCRIPTION

Csh is a command language interpreter. It begins by executing commands from the file '.cshrc' in the home directory of the invoker. If this is a login shell, then it also executes commands from the file '.login' there. In the normal case, the shell will then begin reading commands from the terminal, prompting with '% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

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

When a login shell terminates, it executes commands from the file '.logout' in the users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters '&' '|' ';' '<' '>' '(' ')' form separate words. If doubled in '&&', '||', '<<' or '>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with '\'. A newline preceded by a '\' is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, ''', '`' or '"', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of '' or '"' characters a newline preceded by a '\' gives a true newline character.

When the shell's input is not a terminal, the character '#' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by '\' and in quotations using '`', ''', and '"'.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a

sequence of simple commands separated by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ';', and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an '&'. Such a sequence is automatically prevented from being terminated by a hangup signal; the nohup command need not be used.

Any of the above may be placed in '(' ')' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with '||' or '&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See Expressions.)

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus history substitutions provide a generalization of a redo function.

History substitutions begin with the character '!' and may begin **anywhere** in the input stream if a history substitution is not already in progress. This '!' may be preceded by an '\' to prevent its special meaning; a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. History substitutions also occur when an input line begins with ''. This special abbreviation will be described later.

Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list, the size of which is controlled by the history variable. The previous command is always retained. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the history command:

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

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an '!' in the prompt string.

With the current event 13 we can refer to previous events by event number '!11', relatively as in '!-2' (referring to the same event), by a prefix of a command word as in '!d' for event 12 or '!w' for event 9, or by a string contained in a word in the command as in '!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '!!' refers to the previous command; thus '!!' alone is essentially a redo. The form '!#' references the current command (the one being typed in). It allows a word to be selected from further left in the line, to avoid retyping a long name, as in '!#:1'.

To select words from an event we can follow the event specification by a ':' and a designator for the desired words. The words of a input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, and so on. The basic word designators are:

0	first (command) word
n	n'th argument
	first argument, i.e. '1'
$	last argument
%	word matched by (immediately preceding) ?s? search
x-y	range of words
-y	abbreviates '0-y'
*	abbreviates '-$', or nothing if only 1 word in event
x*	abbreviates 'x-$'
x-	like 'x*' but omitting word '$'

The ':' separating the event specification from the word designator can be omitted if the argument selector begins with a '', '$', '*' '-' or '%'. After the optional word designator can be placed a sequence of modifiers, each preceded by a ':'. The following modifiers are defined:

h	Remove a trailing pathname component.
r	Remove a trailing '.xxx' component.
s/l/r/	Substitute l for r
t	Remove all leading pathname components.
&	Repeat the previous substitution.

g Apply the change globally, prefixing the above.
p Print the new command but do not execute it.
q Quote the substituted words, preventing substitutions.
x Like q, but break into words at blanks, tabs, and newlines.

Unless preceded by a 'g' the modification is applied only to the first modifiable word. In any case it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of '/'; a '\' quotes the delimiter into the l and r strings. The character '&' in the right hand side is replaced by the text from the left. A '\' quotes '&' also. A null l uses the previous string either from a l or from a contextual scan string s in '!?s?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing '?' in a contextual scan.

A history reference may be given without an event specification, e.g. '!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus '!?foo? !$' gives the first and last arguments from the command matching '?foo?'.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a ''. This is equivalent to '!:s' providing a convenient shorthand for substitutions on the text of the previous line. Thus 'lblib' fixes the spelling of 'lib' in the previous command. Finally, a history substitution may be surrounded with '{' and '}' if necessary to insulate it from the characters which follow. Thus, after 'ls -ld ~paul' we might do '!{l}a' to do 'ls -ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ' and "

The quotation of strings by ''' and '"' can be used to prevent all or some of the remaining substitutions. Strings enclosed in ''' are prevented any further interpretation. Strings enclosed in '"' are yet variable and command expanded as described below.

In both cases, the resulting text becomes (all or part of) a single word; only in one special case (see Command Substitition below) does a " quoted string yield parts of more than one word; ' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the alias and unalias commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for 'ls' is 'ls -l' the command 'ls /usr' would map to 'ls -l /usr', the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep !^ /etc/passwd' then 'lookup bill' would map to 'grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can 'alias print 'pr \!* | lpr'' to make a command which pr's its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the argv variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the set and unset commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the verbose variable is a toggle which causes command input to be echoed. The setting of this variable results from the **-v** command line option.

Other operations treat variables numerically. The '@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the

purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '$' characters. This expansion can be prevented by preceding the '$' with a '\' except within '"'s where it **always** occurs, and within ''s where it **never** occurs. Strings quoted by '`' are interpreted later (see Command substitution below) so '$' substitution does not occur there until later, if at all. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.

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

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

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

$name
${name}
Are replaced by the words of the value of variable name, each separated by a blank. Braces insulate name from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If name is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}
May be used to select only some of the words from the value of name. The selector is subjected to '$'

substitution and may consist of a single number or two numbers separated by a '-'. The first word of a variables value is numbered '1'. If the first number of a range is omitted it defaults to '1'. If the last member of a range is omitted it defaults to '$#name'. The selector '*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}
Gives the number of words in the variable. This is useful for later use in a '[selector]'.

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

$number
${number}
Equivalent to '$argv[number]'.

$*
Equivalent to '$argv[*]'.

The modifiers ':h', ':t', ':r', ':q' and ':x' may be applied to the substitutions above as may ':gh', ':gt' and ':gr'. If braces '{' '}' appear in the command form then the modifiers must appear within the braces. **The current implementation allows only one ':' modifier on each '$' expansion.**

The following substitutions may not be modified with ':' modifiers.

$?name
${?name}
Substitutes the string '1' if name is set, '0' if it is not.

$?0
Substitutes '1' if the current input filename is know, '0' if it is not.

$$
Substitute the (decimal) process number of the (parent) shell.

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin

commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in '`'. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within '"'s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters '*', '?', '[' or '{' or begins with the character '~', then that word is a candidate for filename substitution, also known as 'globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters '*', '?' and '[' imply pattern matching, the characters '~' and '{' being more akin to abbreviations.

In matching filenames, the character '.' at the beginning of a filename or immediately following a '/', as well as the character '/' must be matched explicitly. The character '*' matches any string of characters, including the null string. The character '?' matches any single character. The sequence '[...]' matches any one of the characters enclosed. Within '[...]', a pair of characters separated by '-' matches any character lexically between the two.

The character '~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. '~' it expands to the invokers home directory as reflected in the value of the variable home. When followed by a name consisting of letters, digits and '-' characters the shell searches for a user with that name and substitutes their home directory; thus '~ken' might expand to '/usr/ken' and '~ken/chmach' to '/usr/ken/chmach'. If the character '~' is

followed by a character other than a letter or '/' or appears not at the beginning of a word, it is left undisturbed.

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

Input/output

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

< name
: Open file name (which is first variable, command and filename expanded) as the standard input.

<< word
: Read the shell input up to a line which is identical to word. Word is not subjected to variable, filename or command substitution, and each input line is compared to word before any substitutions are done on this input line. Unless a quoting '\', '"', ''' or '`' appears in word variable and command substitution is performed on the intervening lines, allowing '\' to quote '$', '\' and '`'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name
: The file name is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

  If the variable noclobber is set, then the file must not exist or be a character special file (e.g. a terminal or '/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the '!' forms can be used and suppress this check.

The forms involving '&' route the diagnostic output into the specified file as well as the standard output. Name is expanded in the same way as '<' input filenames are.

>> name
>>& name
>>! name
>>&! name
Uses file name as standard output like '>' but places output at the end of the file. If the variable noclobber is set, then it is an error for the file not to exist unless one of the '!' forms is given. Otherwise similar to '>'.

If a command is run detached (followed by '&') then the default standard input for the command is the empty file '/dev/null'. Otherwise the command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The '<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

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

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, exit, if, and while commands. The following operators are available:

|| && | ^ & == != <= >= < > << >> + - * / % ! ~ ( )

Here the precedence increases to the right, '==' and '!=', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '-', '*' '/' and '%' being, in groups, at the same level. The '==' and '!=' operators compare their arguments as strings, all others operate on numbers. Strings which begin with '0' are considered octal numbers. Null or missing arguments are considered '0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the

same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they should be surrounded by spaces.

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

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

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

Control flow

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

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

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

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

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

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

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

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

case label:
A label in a switch statement as discussed below.

cd
cd name
chdir
chdir name
Change the shells working directory to directory name. If no argument is given then change to the home directory of the user.

If name is not found as a subdirectory of the current directory (and does not begin with '/', './', or '../'), then each component of the variable cdpath is checked to see if it has a subdirectory name. Finally, if all else fails but name is a shell variable whose value begins with '/', then this is tried to see if it is a directory.

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

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

echo wordlist
The specified words are written to the shells standard output. A '\c' causes the echo to complete without printing a newline, akin to the '\c' in nroff(1T). A '\n' in wordlist causes a newline to be printed. Otherwise the words are echoed, separated by spaces.

else
end
endif
endsw
See the description of the foreach, if, switch, and while statements below.

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

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

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

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

glob wordlist
Like echo but no '\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word
The specified word is filename and command expanded to

yield a string of the form 'label'. The shell rewinds its input as much as possible and searches for a line of the form 'label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

history
Displays the history event list.

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

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

login
Terminate a login shell, replacing it with an instance of **/bin/login**. This is one way to log off, included for compatibility with **/bin/sh**.

logout
Terminate a login shell. Especially useful if ignoreeof is set.

nice
nice +number
nice command
nice +number command
The first form sets the nice for this shell to 4. The second form sets the nice to the given number. The final two forms run command at priority 4 and number respectively. The super-user may specify negative niceness by using 'nice -number ...'. Command is

always executed in a sub-shell, and the restrictions place on commands in simple if statements apply.

nohup
nohup command
The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. On the Computer Center systems at UC Berkeley, this also submits the process. Unless the shell is running detached, nohup has no effect. All processes detached with "&" are automatically nohup'ed. (Thus, nohup is not really needed.)

onintr
onintr -
onintr label
Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form 'onintr -' causes all interrupts to be ignored. The final form causes the shell to execute a 'goto label' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of onintr have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

rehash
Causes the internal hash table of the contents of the directories in the path variable to be recomputed. This is needed if new commands are added to directories in the path while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command
The specified command which is subject to the same restrictions as the command in the one line if statement above, is executed count times. I/O redirections occurs exactly once, even if count is 0.

set
set name
set name=word
set name[index]=word
set name=(wordlist)
The first form of the command shows the value of all

shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets name to the null string. The third form sets name to the single word. The fourth form sets the index'th component of name to word; this component must already exist. The final form sets name to the list of words in wordlist. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv name value
(Version 7 systems only.) Sets the value of environment variable name to be value, a single string. Useful environment variables are 'TERM' the type of your terminal and 'SHELL' the shell you are using.

shift
shift variable
The members of argv are shifted to the left, discarding argv[1]. It is an error for argv not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name
The shell reads commands from name. Source commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a source at any level terminates all nested source commands. Input during source commands is **never** placed on the history list.

```
switch (string)
case str1:
    ...
  breaksw
...
default:
    ...
  breaksw
endsw
```

Each case label is successively matched, against the specified string which is first command and filename expanded. The file metacharacters '*', '?' and '[...]' may be used in the case labels, which are variable expanded. If none of the labels match before a 'default' label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line.

The command breaksw causes execution to continue after the endsw. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the endsw.

time
time command
With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the time variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask value
The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern
All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by 'unalias *'. It is not an error for nothing to be unaliased.

unhash
Use of the internal hash table to speed location of executed programs is disabled.

unset pattern
All variables whose names match the specified pattern are removed. Thus all variables are removed by 'unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be unset.

wait
All child processes are waited for. It the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (expr)
...
end
While the specified expression evaluates non-zero, the commands between the while and the matching end are evaluated. Break and continue may be used to terminate or continue the loop prematurely. (The while and end

must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the foreach statement if the input is a terminal.

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

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

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

Pre-defined variables

The following variables have special meaning to the shell. Of these, argv, child, home, path, prompt, shell and status are always set by the shell. Except for child and status this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

The shell copies the environment variable PATH into the variable path, and copies the value back into the environment whenever path is set. Thus is is not necessary to worry about its setting other than in the file .cshrc as inferior csh processes will import the definition of path from the environment.

argv Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. '$1' is replaced by '$argv[1]', etc.

cdpath Gives a list of alternate directories searched to find subdirectories in chdir commands.

child The process number printed when the last command was forked with '&'. This variable is unset when this process terminates.

echo Set when the **-x** command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

histchars Can be assigned a two character string. The first character is used as a history character in place of "!", the second character is used in place of the "^" substitution mechanism. For example, "set histchars=",;"" will cause the history characters to be comma and semicolon.

history Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of history may run the shell out of memory. The last executed command is always saved on the history list.

home The home directory of the invoker, initialized from the environment. The filename expansion of '~' refers to this variable.

ignoreeof If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.

mail The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says 'You have new mail.' if the file exists with an access time not greater than its modify time.

If the first word of the value of mail is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

If multiple mail files are specified, then the shell says 'New mail in name' when there is mail in the file name.

noclobber As described in the section on Input/output, restrictions are placed on output redirection

to insure that files are not accidentally destroyed, and that '>>' redirections refer to existing files.

noglob If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. 'echo [' still gives an error.

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no path variable then only full path names will execute. The usual search path is '.', '/bin' and '/usr/bin', but this may vary from system to system. For the super-user the default search path is '/etc', '/bin' and '/usr/bin'. A shell which is given neither the -c nor the -t option will normally hash the contents of the directories in the path variable after reading .cshrc, and each time the path variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the rehash or the commands may not be found.

prompt The string which is printed before each command is read from an interactive terminal input. If a '!' appears in the string it will be replaced by the current event number unless a preceding '\' is given. Default is '% ', or '# ' for the super-user.

shell The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of Non-builtin Command Execution below.) Initialized to the (system-dependent) home of the shell.

status The status returned by the last command. If it terminated abnormally, then 0200 is added

to the status. Builtin commands which fail return exit status '1', all other builtin commands set status '0'.

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the **-v** command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via exec(2). Each word in the variable path names a directory from which the shell will attempt to execute the command. If it is given neither a **-c** nor a -t option, the shell will hash the names in these directories into an internal table so that it will only try an exec in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via unhash), or if the shell was given a -c or -t argument, and in any case for each directory component of path which does not begin with a "/", the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus '(cd ; pwd) ; pwd' prints the home directory; leaving you where you were (printing this after the home directory), while 'cd ; pwd' leaves you in the home directory. Parenthesized commands are most often used to prevent chdir from affecting the current shell.

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

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

Argument list processing

If argument 0 to the shell is '-' then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in argv.

-e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell will start faster, because it will neither search for nor execute commands from the file '.cshrc' in the invokers home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A '\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the verbose variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the echo variable to be set, so that commands are echoed immediately before execution.

-V Causes the verbose variable to be set even before '.cshrc' is executed.

-X Is to **-x** as **-V** is to **-v**.

After processing of flag arguments if arguments remain but none of the -c, -i, -s, or -t options was given the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a 'standard' shell if the first character of a script is not a '#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable

argv.

Signal handling

The shell normally ignores quit signals. The interrupt and quit signals are ignored for an invoked command if the command is followed by '&'; otherwise the signals have the values which the shell inherited from its parent. The shells handling of interrupts can be controlled by onintr. Login shells catch the terminate signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file '.logout'.

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after '.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Shell for scripts not starting with a '#'.
/tmp/sh*	Temporary file for '<<'.
/dev/null	Source of empty file.
/etc/passwd	Source of home directories for '~name'.

LIMITATIONS

Words can be no longer than 512 characters. The number of characters in an argument varies from system to system. Early version 6 systems typically have 512 character limits while later version 6 and version 7 systems have 5120 character limits. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Also command substitutions may substitute no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of alias substititutions on a single line to 20.

SEE ALSO

access(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), a.out(5), environ(5),

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

Commands within loops, prompted for by '?', are not placed in the history list.

It should be possible to use the ':' modifiers on the output of command substitutions. All and more than one ':' modifier should be allowed on '$' substitutions.

Some commands should not touch status or it may be so transient as to be almost useless. Oring in 0200 to status on abnormal termination is a kludge.

In order to be able to recover from failing exec commands on version 6 systems, the new command inherits several open files other than the normal standard input and output and diagnostic output. If the input and output are redirected and the new command does not close these files, some files may be held open unnecessarily.

There are a number of bugs associated with the importing/exporting of the PATH. For example, directories in the path using the ~ syntax are not expanded in the PATH. Unusual paths, such as (), can cause csh to core dump.

This version of csh does not support or use the process control features of the 4th Berkeley Distribution. It contains a number of known bugs which have been fixed in the process control version. This version is not supported.

NAME

ctags - create a tags file

SYNTAX

ctags [-u] [-w] [-x] name ...

DESCRIPTION

Ctags makes a tags file for vi(1) from the specified C, Pascal and Fortran sources. A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the tags file, vi can quickly find these function definitions.

If the **-x** flag is given, ctags produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. This is a simple index which can be printed out as an off-line readable function index.

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. Others are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are processed again looking for C definitions.

Other options are:

-w suppressing warning diagnostics.

-u causing the specified files to be updated in tags, that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the tags file.)

The tag main is treated specially in C programs. The tag formed is created by prepending M to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of ctags practical in directories with more than one program.

FILES

tags output tags file

SEE ALSO

ex(1), vi(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name you lose.

NAME

cu - call up XENIX

SYNTAX

cu telno [-t] [-s speed] [-a acu] [-l line] [-nh]

DESCRIPTION

Cu calls up another XENIX system, a terminal, or possibly a non-XENIX system. It manages an interactive conversation with possible transfers of text files. Telno is the telephone number, with minus signs at appropriate places for delays, or 'wait', to indicate a manual connection. If 'wait' is specified, '/dev/null' is used as the dial unit and cu waits up to five minutes for the carrier to turn on. The -t flag is used to dial out to a terminal. Speed gives the transmission speed (110, 134, 150, 300, 600, 1200, 2400, 4800, 9600); 300 is the default value. The **-nh** flag prevents cu from hanging up the terminal line upon exit.

The **-a** and **-l** values may be used to specify pathnames for the ACU and communications line devices. They can be used to override the following built-in choices:

-a /dev/cua0 -l /dev/cul0

After making the connection, cu runs as two processes: the send process reads the standard input and passes most of it to the remote system; the receive process reads from the remote system and passes most data to the standard output. Lines beginning with '~' have special meanings.

The send process interprets the following:

~.	terminate the conversation.
~EOT	terminate the conversation
~<file	send the contents of file to the remote system, as though typed at the terminal.
~!	invoke an interactive shell on the local system.
~!cmd ...	run the command on the local system (via **sh -c**).
~$cmd ...	run the command locally and send its output to the remote system.
~%take from [to]	copy file 'from' (on the remote system) to file 'to' on the local system. If 'to' is omitted, the 'from' name is used both places.

~%put from [to]	copy file 'from' (on local system) to file 'to' on remote system. If 'to' is omitted, the 'from' name is used both places.
~%speed n	set speed of transmission line to 'n', where n is one of 110, 134, 150, 300, 600, 1200, 2400, 4800, 9600.
~~...	send the line '~...'.

The receive process handles output diversions of the following form:

```
~>[>][:]file
zero or more lines to be written to file
~>
```

In any case, output is diverted (or appended, if '>>' used) to the file. If ':' is used, the diversion is silent, i.e., it is written only to the file. If ':' is omitted, output is written both to the file and to the standard output. The trailing '~>' terminates the diversion.

The use of ~%put requires stty and cat on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of ~%take requires the existence of echo and tee on the remote system. Also, stty tabs mode is required on the remote system if tabs are to be copied without expansion.

FILES

/dev/cua0
/dev/cul0
/dev/null

SEE ALSO

dn(4), tty(4)

DIAGNOSTICS

Exit code is zero for normal exit, nonzero (various values) otherwise.

NOTES

The syntax is unique.

NAME

date - print and set the date

SYNTAX

date [yymmddhhmm [.ss]]

DESCRIPTION

If no argument is given, the current date and time are printed. If an argument is given, the current date is set. yy is the last two digits of the year; the first mm is the month number; dd is the day number in the month; hh is the hour number (24 hour system); the second mm is the minute number; .ss is optional and is the seconds. For example:

```
date 10080045
```

sets the date to Oct 8, 12:45 AM. The year, month and day may be omitted, the current values being the defaults. The system operates in GMT. Date takes care of the conversion to and from local standard and daylight time.

FILES

/usr/adm/wtmp to record time-setting

SEE ALSO

utmp(5)

DIAGNOSTICS

'No permission' if you aren't the super-user and you try to change the date; 'bad conversion' if the date set is syntactically incorrect.

NAME

dc - desk calculator

SYNTAX

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of dc is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number

The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.

+ - / * % ^

The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the **l** is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged. **P** interprets the top of the stack as an ascii string, removes it, and prints it.

f All values on the stack and in registers are printed.

q exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x treats the top element of the stack as a character string and executes it as a string of dc commands.

X replaces the number on the top of the stack with its scale factor.

[...]
puts the bracketed ascii string onto the top of the stack.

<x >x =x
The top two elements of the stack are popped and compared. Register x is executed if they obey the stated relation.

v replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! interprets the rest of the line as a XENIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. **I** pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O pushes the output base on the top of the stack.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : are used by bc for array operations.

An example which prints the first ten values of n! is

```
[lal+dsa*plal0>y]sy
0sal
lyx
```

SEE ALSO

bc(1), which is a preprocessor for dc providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

'x is unimplemented' where x is an octal number.
'stack empty' for not enough elements on the stack to do what was asked.
'Out of space' when the free list is exhausted (too many digits).
'Out of headers' for too many numbers being kept around.
'Out of pushdown' for too many items on the stack.
'Nesting Depth' for too many levels of nested execution.

NAME

dcheck - file system directory consistency check

SYNTAX

dcheck [**-i** numbers] [filesystem]

DESCRIPTION

Dcheck reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The **-i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

FILES

Default file systems vary with installation.

SEE ALSO

icheck(1), filsys(5), clri(1), ncheck(1)

DIAGNOSTICS

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

NOTES

Since dcheck is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

Fsck is in most cases a better utility for repairing file systems, and also requires somewhat less technical knowledge than dcheck to use effectively.

NAME

dd - convert and copy a file

SYNTAX

dd [option=value] ...

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

option	values
if=	input file name; standard input is default
of=	output file name; standard output is default
ibs=n	input block size n bytes (default 512)
obs=n	output block size (default 512)
bs=n	set both input and output block size, superseding ibs and obs; also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=n	conversion buffer size
skip=n	skip n input records before starting copy
files=n	copy n files from (tape) input
seek=n	seek n records from beginning of output file before copying
count=n	copy only n input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to ibs
... , ...	several comma-separated conversions

Where sizes are specified, a number of bytes is expected. A number may end with **k**, **b** or **w** to specify multiplication by 1024, 512, or 2 respectively; a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if ascii or ebcdic conversion is specified. In the former case cbs characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and new-line added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size cbs.

After completion, dd reports the number of whole and partial input and output blocks.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file x:

```
dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. Dd is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

To skip over a file before copying from magnetic tape do

```
(dd of=/dev/null; dd of=x) </dev/rmt0
```

SEE ALSO

cp(1), tr(1)

DIAGNOSTICS

f+p records in(out): numbers of full and partial records read(written)

NOTES

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. The 'ibm' conversion corresponds better to certain IBM print train conventions.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

NAME

deroff - remove nroff, troff, tbl and eqn constructs

SYNTAX

deroff [-w] file ...

DESCRIPTION

Deroff reads each file in sequence and removes all nroff(1T) and troff(1T) command lines, backslash constructions, macro definitions, eqn constructs (between '.EQ' and '.EN' lines or between delimiters), and table descriptions and writes the remainder on the standard output. Deroff follows chains of included files ('.so' and '.nx' commands); if a file has already been included, a '.so' is ignored and a '.nx' terminates execution. If no input file is given, deroff reads from the standard input file.

If the -w flag is given, the output is a word list, one 'word' (string of letters, digits, and apostrophes, beginning with a letter; apostrophes are removed) per line, and all other characters ignored. Otherwise, the output follows the original, with the deletions mentioned above.

SEE ALSO

nroff(1T), troff(1T), eqn(1T), tbl(1T)

NOTES

Deroff is not a complete troff interpreter, so it can be confused by subtle constructs. Most errors result in too much rather than too little output.

NAME

df - disk free

SYNTAX

df [filesystem] ...

DESCRIPTION

Df prints out the number of free blocks available on the filesystems. If no file system is specified, the free space on all of the normally mounted file systems is printed.

FILES

Default file systems vary with installation.

SEE ALSO

icheck(1)

NAME

diff - differential file comparator

SYNTAX

diff [**-efbh**] file1 file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If file1 (file2) is '-', the standard input is used. If file1 (file2) is a directory, then a file in that directory whose file-name is the same as the file-name of file2 (file1) is used. The normal output contains lines of these forms:

n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4

These lines resemble ed commands to convert file1 into file2. The numbers after the letters pertain to file2. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert file2 into file1. As in ed, identical pairs where n1 = n2 or n3 = n4 are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

The **-b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The **-e** option produces a script of a, c and d commands for the editor ed, which will recreate file2 from file1. The **-f** option produces a similar script, not useful with ed, in the opposite order. In connection with **-e**, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version ed scripts ($2,$3,...) made by diff need be on hand. A 'latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Except in rare circumstances, diff finds a smallest sufficient set of file differences.

Option **-h** does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options **-e** and **-f** are unavailable with **-h**.

FILES

/tmp/d?????
/usr/lib/diffh for **-h**

SEE ALSO

cmp(1), comm(1), ed(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

NOTES

Editing scripts produced under the **-e** or **-f** option are naive about creating lines consisting of a single '.'.

NAME

diff3 - 3-way differential file comparison

SYNTAX

diff3 [**-ex3**] filel file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	filel is different
====2	file2 is different
====3	file3 is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : nl a	Text is to be appended after line number nl in file f, where f = 1, 2, or 3.
f : nl , n2 c	Text is to be changed in the range line nl to line n2. If nl = n2, the range may be abbreviated to nl.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **-e** option, diff3 publishes a script for the editor ed that will incorporate into filel all changes between file2 and file3, i.e. the changes that normally would be flagged ==== and ====3. Option **-x** (-3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to 'filel'.

```
(cat script; echo '1,$p') | ed - filel
```

FILES

/tmp/d3?????
/usr/lib/diff3

SEE ALSO

diff(1)

NOTES

Text lines that consist of a single '.' will defeat -e.
Files longer than 64K bytes won't work.

NAME

disable - turn off terminals

SYNTAX

disable ttyn ...

DESCRIPTION

This program manipulates the /etc/ttys file and signals init to disallow logins on a particular terminal. The user must be in the same group as the user root.

Normally disable will disallow logins on the terminals specified.

FILES

/dev/tty*, /etc/ttys

SEE ALSO

login(1), enable(1), ttys(5), getty(8), init(8)

WARNING

Be absolutely certain to pause at least one minute before reusing this command or before using the enable command. Failure to do so may cause the system to crash.

NAME

du - summarize disk usage

SYNTAX

du [-s] [-a] [name ...]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each specified directory or file name. If name is missing, '.' is used.

The optional argument -s causes only the grand total to be given. The optional argument -a causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

A file which has two links to it is only counted once.

NOTES

Non-directories given as arguments (not under -a option) are not listed.
If there are too many distinct linked files, du counts the excess files multiply.

NAME

dump - incremental file system dump

SYNTAX

dump [key [argument ...] filesystem]

DESCRIPTION

Dump copies to magnetic tape all files changed after a certain date in the filesystem. The key specifies the date and other options about the dump. The key consists of characters from the set **0123456789fusd.**

f Place the dump on the next argument file instead of the tape.

u If the dump completes successfully, write the date of the beginning of the dump on file '/etc/ddate'. This file records a separate date for each filesystem and each dump level.

0-9 This number is the 'dump level'. All files modified since the last date stored in the file '/etc/ddate' for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option 0 causes the entire filesystem to be dumped.

s The size of the dump tape is specified in feet. The number of feet is taken from the next argument. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2300 feet.

d The density of the tape, expressed in BPI, is taken from the next argument. This is used in calculating the amount of tape used per write. The default is 1600.

k This option is used when dumping to a block-structured device, such as a floppy disk. The size (in K-bytes) of the volume being written is taken from the next argument. If the k argument is specified, any s and d arguments are ignored. The default is to use s and d.

If no arguments are given, the key is assumed to be **9u** and the program attempts to dump the default filesystem to the default tape.

Now a short suggestion on how perform dumps. Start with a full level 0 dump

dump 0u

Next, periodic level 9 dumps should be made on an exponential progression of tapes. (Sometimes called Tower of Hanoi - 1 2 1 3 1 2 1 4 ... tape 1 used every other time, tape 2 used every fourth, tape 3 used every eighth, etc.)

```
dump 9u
```

When the level 9 incremental approaches a full tape (about 78000 blocks at 1600 BPI blocked 20), a level 1 dump should be made.

```
dump 1u
```

After this, the exponential series should progress as uninterrupted. These level 9 dumps are based on the level 1 dump which is based on the level 0 full dump. This progression of levels of dump can be carried as far as desired.

FILES

Default filesystem and tape vary with installation. For safety, however, we recommend that default disk filesystems not be used, as common operator errors can destroy that default disk.
/etc/ddate: record dump dates of filesystem/level.

SEE ALSO

restor(1), dump(5), dumpdir(1)

DIAGNOSTICS

If the dump requires more than one tape, it will ask you to change tapes. Reply with a new-line when this has been done.

NOTES

Sizes are based on 1600 BPI blocked tape. The raw magtape device has to be used to approach these densities. Read errors on the filesystem are ignored. Write errors on the magtape are usually fatal.

NAME

dumpdir - print the names of files on a dump tape

SYNTAX

dumpdir [f filename]

DESCRIPTION

Dumpdir is used to read magtapes dumped with the dump command and list the names and inode numbers of all the files and directories on the tape.

The f option causes filename as the name of the tape instead of the default.

FILES

default tape unit varies with installation
rst*

SEE ALSO

dump(1), restor(1)

DIAGNOSTICS

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

NOTES

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, dumpdir doesn't use it.

NAME

echo - echo arguments

SYNTAX

echo [-n] [arg] ...

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a newline on the standard output. If the flag -n is used, no newline is added to the output.

Echo is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send diagnostics to the standard error file, do 'echo ... 1>&2'.

NAME

ed - text editor

SYNTAX

ed [-] [-x] [name]

DESCRIPTION

Ed is the standard text editor.

If a name argument is given, ed simulates an e command (see below) on the named file; that is to say, the file is read into ed's buffer so that it can be edited. If -x is present, an x command is simulated first to handle an encrypted file. The optional - suppresses the printing of character counts by e, r, and w commands.

Ed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a w (write) command is given. The copy of the text being edited resides in a temporary file called the buffer.

Commands to ed have a simple and regular structure: zero or more addresses followed by a single character command, possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Missing addresses are supplied by default.

In general, only one command may appear on a line. Certain commands allow the addition of text to the buffer. While ed is accepting text, it is said to be in input mode. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period '.' alone at the beginning of a line.

Ed supports a limited form of regular expression notation. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. In the following specification for regular expressions the word 'character' means any character but newline.

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus \[. and sometimes ^*$.

2. A . matches any character.

3. A \ followed by any character except a digit or () matches that character.

4. A nonempty string s bracketed [s] (or [^s]) matches any character in (or not in) s. In s, \ has no special

meaning, and] may only appear as the first letter. A substring a-b, with a and b in ascending ASCII order, stands for the inclusive range of ASCII characters.

5. A regular expression of form 1-4 followed by * matches a sequence of 0 or more matches of the regular expression.

6. A regular expression, x, of form 1-8, bracketed \(x\) matches what x matches.

7. A \ followed by a digit n matches a copy of the string that the bracketed regular expression beginning with the nth \(matched.

8. A regular expression of form 1-8, x, followed by a regular expression of form 1-7, y matches a match for x followed by a match for y, with the x match being as long as possible while still permitting a y match.

9. A regular expression of form 1-8 preceded by ^ (or followed by $), is constrained to matches that begin at the left (or end at the right) end of a line.

10. A regular expression of form 1-9 picks out the longest among the leftmost matches in a line.

11. An empty regular expression stands for a copy of the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see s below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by '\'. This also applies to the character bounding the regular expression (often '/') and to '\' itself.

To understand addressing in ed it is necessary to know that at any time there is a current line. Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character '.' addresses the current line.

2. The character '$' addresses the last line of the buffer.

3. A decimal number n addresses the n-th line of the buffer.

4. ''x' addresses the line marked with the name x, which must be a lower-case letter. Lines are marked with the k command described below.

5. A regular expression enclosed in slashes '/' addresses the line found by searching forward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the beginning of the buffer.

6. A regular expression enclosed in queries '?' addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the end of the buffer.

7. An address followed by a plus sign '+' or a minus sign '-' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.

8. If an address begins with '+' or '-' the addition or subtraction is taken with respect to the current line; e.g. '-5' is understood to mean '.-5'.

9. If an address ends with '+' or '-', then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address '-' refers to the line before the current line. Moreover, trailing '+' and '-' characters have cumulative effect, so '--' refers to the current line less 2.

10. To maintain compatibility with earlier versions of the editor, the character '^' in addresses is equivalent to '-'.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma ','. They may also be separated by a semicolon ';'. In this case the current line '.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/', '?'). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of ed commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, most commands may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed respectively in the way discussed below.

(.)a
<text>
.

The append command reads the given text and appends it after the addressed line. '.' is left on the last line input, if there were any, otherwise at the addressed line. Address '0' is legal for this command; text is placed at the beginning of the buffer.

(., .)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. '.' is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

(., .)d

The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. '.' is set to the last line of the buffer. The number of characters read is typed. 'filename' is remembered for possible use as a default file name in a subsequent r or w command. If 'filename' is missing, the remembered name is used.

E filename

This command is the same as e, except that no diagnostic results when no w has been given since the last buffer alteration.

f filename

The filename command prints the currently remembered file name. If 'filename' is given, the currently

remembered file name is changed to 'filename'.

(1,$)g/regular expression/command list
In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with '.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with '\'. A, i, and c commands and associated input are permitted; the '.' terminating input mode may be omitted if it would be on the last line of the command list. The commands g and v are not permitted in the command list.

(.)i

<text>
.
This command inserts the given text before the addressed line. '.' is left at the last line input, or, if there were none, at the line before the addressed line. This command differs from the a command only in the placement of the text.

(., .+1)j
This command joins the addressed lines into a single line; intermediate newlines simply disappear. '.' is left at the resulting line.

(.)kx
The mark command marks the addressed line with name x, which must be a lower-case letter. The address form ''x' then addresses this line.

(., .)l
The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in two-digit octal, and long lines are folded. The l command may be placed on the same line after any non-i/o command.

(., .)ma
The move command repositions the addressed lines after the line addressed by a. The last of the moved lines becomes the current line.

(., .)p
The print command prints the addressed lines. '.' is left at the last line printed. The p command may be placed on the same line after any non-i/o command.

(., .)P
This command is a synonym for p.

q The quit command causes ed to exit. No automatic write of a file is done.

Q This command is the same as q, except that no diagnostic results when no w has been given since the last buffer alteration.

($)r filename
The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see e and f commands). The file name is remembered if there was no remembered file name already. Address '0' is legal for r and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. '.' is left at the last line read in from the file.

(., .)s/regular expression/replacement/ or,
(., .)s/regular expression/replacement/g
The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator 'g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of '/' to delimit the regular expression and the replacement. '.' is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of '&' in this context may be suppressed by preceding it by '\'. The characters '\n' where n is a digit, are replaced by the text matched by the n-th regular subexpression enclosed between '\(' and '\)'. When nested, parenthesized subexpressions are present, n is determined by counting occurrences of '\(' starting from the left.

Lines may be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by '\'.

(., .)ta
This command acts just like the m command, except that

a copy of the addressed lines is placed after address a (which may be 0). '.' is left on the last line of the copy.

(., .)u
The undo command restores the preceding contents of the current line, which must be the last line in which a substitution was made.

(1, $)v/regular expression/command list
This command is the same as the global command g except that the command list is executed g with '.' initially set to every line except those matching the regular expression.

(1, $)w filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writable by everyone). The file name is remembered if there was no remembered file name already. If no file name is given, the remembered file name, if any, is used (see e and f commands). '.' is unchanged. If the command is successful, the number of characters written is printed.

(1,$)W filename
This command is the same as w, except that the addressed lines are appended to the file.

x A key string is demanded from the standard input. Later r, e and w commands will encrypt and decrypt the text with this key by the algorithm of crypt(1). An explicitly empty key turns off encryption.

($)= The line number of the addressed line is typed. '.' is unchanged by this command.

!<shell command>
The remainder of the line after the '!' is sent to sh(1) to be interpreted as a command. '.' is unchanged.

(.+1)<newline>
An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1p'; it is useful for stepping through text.

If an interrupt signal (ASCII DEL) is sent, ed prints a '?' and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name,

and 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 1 word.

When reading a file, ed discards ASCII NUL characters and all characters after the last newline. It refuses to read files containing non-ASCII characters.

FILES

/tmp/e*
ed.hup: work is saved here if terminal hangs up

SEE ALSO

sed(1), crypt(1)

DIAGNOSTICS

'?name' for inaccessible file; '?' for errors in commands; '?TMP' for temporary file overflow.

To protect against throwing away valuable work, a q or e command is considered to be in error, unless a w has occurred since the last buffer change. A second q or e will be obeyed regardless.

NOTES

The l command mishandles DEL.
A ! command cannot be subject to a g command.
Because 0 is an illegal address for a w command, it is not possible to create an empty file with ed.

NAME

egrep - search a file for a full regular expression

SYNTAX

egrep [option] ... [expression] [file] ...

DESCRIPTION

Commands of the grep family search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The file name is shown if there is more than one input file.

Egrep patterns are full regular expressions. The algorithm used is a fast deterministic algorithm that sometimes needs exponential space. Normally, egrep is used when searching for complicated regular expressions. Use fgrep to find fixed strings, and use grep to find limited regular expressions consistent with the syntax used by ed and ex.

The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its line number in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s No output is produced, but the correct exit status is returned. (See DIAGNOSTICS below.) Useful for detecting a given pattern in a file before performing an operation on it.

-h Do not print filename headers with output lines.

-e expression
Same as a simple expression argument, but useful when the expression begins with a -.

-f file
The regular expression is taken from file.

Care should be taken when using the characters $ * [^ | ? ' " ( ) and \ in an expression as they are also meaningful to the shell. It is safest to enclose the entire expression argument in single quotes ' '.

Egrep accepts extended regular expressions consisting of the following elements. Note that in the following description 'character' excludes newline:

A backslash (\) followed by a single character matches that character. This is true for all of the characters discussed below, so that the special meaning of any character ceases if preceded by a backslash.

A caret (^) matches the beginning of a line.

A dollar sign ($) matches the end of a line.

A period (.) matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in "a-z0-9". An end bracket (]) may occur only as the first character of the string. A literal hyphen (-) must be placed where it can't be mistaken as a range indicator.

A regular expression followed by an asterisk (*) matches a sequence of 0 or more matches of the regular expression.

A regular expression followed by a plus sign (+) matches a sequence of 1 or more matches of the regular expression.

A regular expression followed by a question mark (?) matches a sequence of 0 or 1 matches of the regular expression.

Two concatenated regular expressions are the same as a match of the first followed by a match of the second.

Two regular expressions separated by a vertical bar (|) or newline signify a match of just one of the two regular expressions.

A regular expression enclosed in parentheses is the same as a match for the enclosed regular expression.

The order of precedence for operators at the same parenthesis level is brackets ([]), then character expansion and repetition (* ,+ , and ?), followed by concatenation, and then selection (| and newline).

EXAMPLES

All examples below find regular expressions in the files names filel and file2 .

Find all lines beginning with an upper case letter:

```
egrep '^[A-Z]' filel file2
```

Find all lines containing an asterisk:

```
egrep '\*' filel file2
```

Find all lines ending with a backslash:

```
egrep '\\$' filel file2
```

Find all lines not beginning with ".H":

```
egrep -v '^\.H' filel file2
```

Find all lines containing exactly three characters:

```
egrep '...' filel file2
```

Find all lines containing any of the regular expressions listed in expressfile :

```
egrep -f expressfile filel file2
```

SEE ALSO

ed(1), fgrep(1), grep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

NOTES

Ideally there should be only one grep, but no know single algorithm exists that spans a wide enough range of space-time tradeoffs.

Lines are limited to 256 characters; longer lines are truncated.

NAME

enable - turn on terminals

SYNTAX

enable [**-ed**] tty*n* [[**-ed**] tty*n*] ...

DESCRIPTION

This program manipulates the /etc/ttys file and signal *init* to allow logins on a particular terminal. The user must be in the same group as the user *root*.

Normally *enable* will allow logins on the specified terminals. The **-e** and **-d** flags may be used to allow logins on some terminals and disallow logins on other terminals in a single command.

FILES

/dev/tty*, /etc/ttys

SEE ALSO

login(1), disable(1), ttys(5), getty(8), init(8)

WARNING

Be absolutely certain to pause at least **one minute** before reusing this command or before using the *disable* command. Failure to do so may cause the system to crash.

NAME

eqn, - typeset mathematics

SYNTAX

eqn [-dxy] [-pn] [-sn] [-fn] [file] ...

DESCRIPTION

Eqn is a troff(1T) preprocessor for typesetting mathematics on a Graphic Systems phototypesetter; neqn is its counterpart for nroff(1T) terminals and printers. Usage is almost always

```
eqn file ... | troff
```

If no files are specified, these programs reads from the standard input. A line beginning with '.EQ' marks the start of an equation; the end of an equation is marked by a line beginning with '.EN'. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to set two characters as 'delimiters'; subsequent text between delimiters is also treated as eqn input. Delimiters may be set to characters x and y with the command-line argument -dxy or (more commonly) with 'delim xy' between .EQ and .EN. The left and right delimiters may be identical. Delimiters are turned off by 'delim off'. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

The program checkeq reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within eqn are separated by spaces, tabs, newlines, braces, double quotes, tildes or circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character like x could appear, a complicated construction enclosed in braces may be used instead. Tilde ~ represents a full space in the output, circumflex ^ half as much.

SEE ALSO

neqn(1T), checkeq(1T), troff(1T), tbl(1T), ms(7), eqnchar(7)
B. W. Kernighan and L. L. Cherry, Typesetting Mathematics-User's Guide
J. F. Ossanna, NROFF/TROFF User's Manual

NOTES

To embolden digits, parens, etc., it is necessary to quote them, as in 'bold "12.3"'.

NAME

ex - text editor

SYNTAX

ex [-] [**-v**] [-t tag] [-r] [+lineno] name ...

DESCRIPTION

Ex is the root of a family of editors: edit, ex and vi. Ex is a superset of ed, with the most notable extension being a display editing facility. Display based editing is the focus of vi.

If you have not used ed, or are a casual user, you will find that the editor edit is convenient for you. It avoids some of the complexities of ex used mostly by systems programmers and persons very familiar with ed.

If you have a CRT terminal, you may wish to use a display based editor; in this case see vi(1), which is a command which focuses on the display editing portion of ex.

DOCUMENTATION

For edit and ex see the Ex/edit command summary - Version 2.0. The document Edit: A tutorial provides a comprehensive introduction to edit assuming no previous knowledge of computers or the XENIX system.

The Ex Reference Manual - Version 2.0 is a comprehensive and complete manual for the command mode features of ex, but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of ex see the editing documents written by Brian Kernighan for the editor ed; the material in the introductory and advanced documents works also with ex.

An Introduction to Display Editing with Vi introduces the display editor vi and provides reference material on vi. The Vi Quick Reference card summarizes the commands of vi in a useful, functional way, and is useful with the Introduction.

FOR ED USERS

If you have used ed you will find that ex has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with vi. Generally, the editor uses far more of the capabilities of terminals than ed does, and uses the terminal capability data base termcap(5) and the type of the terminal you are using from the variable TERM in the environment to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its **visual** command (which can be abbreviated **vi**) and which is the central mode of editing when using vi(1).

There is also an interline editing **open** (**o**) command which works on all terminals.

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Hitting ^D causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just hitting return. Of course, the screen oriented **visual** mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (**u**) command allows you to reverse any single change which goes astray. Ex gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files unless you edited them so that you don't accidentally clobber with a write a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the editor **recover** command to retrieve your work. This will get you back to within a few lines of where you left off.

Ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (**n**) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, filenames in the editor may be formed with full shell metasyntax. The metacharacter '%' is also available in forming filenames and is replaced by the name of the current file. For editing large groups of related files you can use ex's tag command to quickly locate functions and other important points in any of the files. This is useful when working on a large program when you want to quickly find the definition of a particular function. The command ctags(1) builds a tags file or a group of C programs.

For moving text between files and within a file the editor has a group of buffers, named a through z. You can place text in these named buffers and carry it over when you edit another file.

There is a command **&** in ex which repeats the last **substitute** command. In addition there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

You can use the **substitute** command in ex to systematically convert the case of letters between upper and lower case. It is possible to ignore case of letters in searches and substitutions. Ex also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word "edit" if your document also contains the word "editor."

Ex has a set of options which you can set to tailor it to your liking. One option which is very useful is the autoindent option which allows the editor to automatically supply leading white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new useful features include an intelligent **join** (j) command which supplies white space between joined lines automatically, commands < and > which shift groups of lines, and the ability to filter portions of the buffer through commands such as sort.

FILES

/usr/lib/ex2.0strings	error messages
/usr/lib/ex2.0recover	recover command
/usr/lib/ex2.0preserve	preserve command
/etc/termcap	describes capabilities of terminals
$HOME/.exrc	editor startup file
/tmp/Exnnnnn	editor temporary
/tmp/Rxnnnnn	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), termcap(5), vi(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

The undo command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The z command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line '-' option is used.

There is no easy way to do a single scan ignoring case.

Because of the implementation of the arguments to next, only 512 bytes of argument list are allowed there.

The format of /etc/termcap and the large number of capabilities of terminals used by the editor cause terminal type setup to be rather slow.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files and cannot appear in resultant files.

NAME

expr - evaluate arguments as an expression

SYNTAX

expr arg ...

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Each token of the expression is a separate argument.

The operators and keywords are listed below. The list is in order of increasing precedence, with equal precedence operators grouped.

expr | expr
yields the first expr if it is neither null nor '0', otherwise yields the second expr.

expr & expr
yields the first expr if neither expr is null or '0', otherwise yields '0'.

expr relop expr
where relop is one of < <= = != >= >, yields '1' if the indicated comparison is true, '0' if false. The comparison is numeric if both expr are integers, otherwise lexicographic.

expr + expr
expr - expr
addition or subtraction of the arguments.

expr * expr
expr / expr
expr % expr
multiplication, division, or remainder of the arguments.

expr : expr
The matching operator compares the string first argument with the regular expression second argument; regular expression syntax is the same as that of ed(1). The \(...\) pattern symbols can be used to select a portion of the first argument. Otherwise, the matching operator yields the number of characters matched ('0' on failure).

(expr)
parentheses for grouping.

Examples:

To add 1 to the Shell variable a:

```
a=`expr $a + 1`
```

To find the filename part (least significant part) of the pathname stored in variable a, which may or may not contain '/':

```
expr $a : '.*/\(.*\)' '|' $a
```

Note the quoted Shell metacharacters.

SEE ALSO

ed(1), sh(1), test(1)

DIAGNOSTICS

Expr returns the following exit codes:

0	if the expression is neither null nor '0',
1	if the expression is null or '0',
2	for invalid expressions.

NAME

false - return false

SYNTAX

false

DESCRIPTION

False does nothing except return with a non-zero exit value. True(1), false 's counterpart, does nothing except return with a zero exit value. False is typically used in shell procedures such as:

```
while test '!false'
do
     command
done
```

SEE ALSO

sh(1) true(1)

DIAGNOSTICS

False has exit status 1.

NAME

fgrep - search a file for a string

SYNTAX

fgrep [option] ... [strings] [file]

DESCRIPTION

Commands of the grep family search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The file name is shown if there is more than one input file.

Fgrep patterns are fixed strings and not regular expressions. Use grep or fgrep when searching for more complicated text patterns. For simple strings, fgrep is faster and more efficient.

The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its linenumber in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s No output is produced, but the correct exit status is returned. (See DIAGNOSTICS below.) Useful for detecting a given pattern in a file before performing an operation on it.

-h Do not print filename headers with output lines.

-y Alphabetic letters in the pattern will match letters of either case in the input.

-e expression
Same as a simple string expression argument, but useful when the string expression begins with a hyphen '-'.

-f file
The string list is taken from file. Fgrep searches for lines that contain one of the (newline-separated) strings.

-x (Exact) only lines matched in their entirety are

printed

Care should be taken when using the characters $ * [^ | ? ' " ( ) and \ in the expression as they are also meaningful to the Shell. In general, it is safest to enclose the entire expression argument in single quotes ' '.

EXAMPLES

All examples below find regular expressions in the files names file1 and file2 .

Find all lines containing an asterisk:

```
fgrep '*' file1 file2
```

Find all lines containing the word "dog":

```
fgrep dog file1 file2
```

Find all lines not containing the letter 'e':

```
fgrep -v e file1 file2
```

Find all lines containing any of the strings listed in stringfile :

```
fgrep -f stringfile file1 file2
```

SEE ALSO

awk(1), ed(1), egrep(1), grep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

NOTES

Ideally there should be only one grep command, but there is no known single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to 256 characters; longer lines are truncated.

NAME

file - determine file type

SYNTAX

file filename ...

file -f fileofnames

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If a file appears to be ascii, file examines the first 512 bytes and tries to guess its language.

If the first argument is a -f flag, file will take the list of filenames from 'fileofnames'.

Several object file formats are recognized. For a.out and x.out format object files, the relationship of cc flags to file classification follows:

cc flag	classification
i	separate
n	pure
s	not "not stripped"

NOTES

It often makes mistakes. In particular it often suggests that command files are C programs.

NAME

find - find files

SYNTAX

find pathname-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each pathname in the pathname-list (i.e., one or more pathnames) seeking files that match a boolean expression written in the primaries given below. In the descriptions, the argument n is used as a decimal integer where +n means more than n, -n means less than n and n means exactly n.

-name filename
: True if the filename argument matches the current file name. Normal Shell argument syntax may be used if escaped (watch out for '[', '?' and '*').

-perm onum
: True if the file permission flags exactly match the octal number onum (see chmod(1)). If onum is prefixed by a minus sign, more flag bits (017777, see stat(2)) become significant and the flags are compared: (flags&onum)==onum.

-type c
: True if the type of the file is c, where c is b, c, d or f for block special file, character special file, directory or plain file.

-links n
: True if the file has n links.

-user uname
: True if the file belongs to the user uname (login name or numeric user ID).

-group gname
: True if the file belongs to group gname (group name or numeric group ID).

-size n
: True if the file is n blocks long (512 bytes per block).

-inum n
: True if the file has inode number n.

-atime n
: True if the file has been accessed in n days.

-mtime n
: True if the file has been modified in n days.

-exec command
: True if the executed command returns a zero value as exit status. The end of the command must be

punctuated by an escaped semicolon. A command argument '{}' is replaced by the current pathname.

-ok command
Like -exec except that the generated command is written on the standard output, then the standard input is read and the command executed only upon response y.

-print Always true; causes the current pathname to be printed.

-access name [rwx]
True if the name has the exact matching permissions to the file. The name may be either a valid login name or a group name. The permission request is optional; it can consist of any of the read, write, or execute combinations. If omitted, it tests whether the directories leading to the file can be searched and the file exists.

-newer file
True if the current file has been modified more recently than the argument file.

The primaries may be combined using the following operators (in order of decreasing precedence):

1) A parenthesized group of primaries and operators (parentheses are special to the Shell and must be escaped).

2) The negation of a primary ('!' is the unary not operator).

3) Concatenation of primaries (the and operation is implied by the juxtaposition of two primaries).

4) Alternation of primaries ('-o' is the or operator).

EXAMPLE

To remove all files named 'a.out' or '*.o' that have not been accessed for a week:

```
find / \( -name a.out -o -name '*.o' \) -atime +7 -exec rm
{} \;
```

FILES

/etc/passwd
/etc/group

SEE ALSO

sh(1), test(1), filsys(5)

NAME

finger - user information lookup program

SYNTAX

finger [options] name ...

DESCRIPTION

By default finger lists the login name, full name, terminal name and write status (as a '*' before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current XENIX user. (Idle time is minutes if it is a single integer, hours and minutes if a colon (:) is present, or days and hours if a 'd' is present.)

A longer format also exists and is used by finger whenever a list of peoples names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line; it includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file .plan in their home directory, and the project on which they are working from the file .project also in the home directory.

Finger options include:

-f Suppress the printing of the header line (short format).

-l Force long output format.

-p Suppress printing of the .plan files

-s Force short output format.

FILES

/etc/utmp	who file
/etc/passwd	user names, offices, phones, login directories, and shells
/usr/adm/lastlog	last login times
$HOME/.plan	plans
$HOME/.project	projects

SEE ALSO

who(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

Only the first line of the .project file is printed.

NAME

fsck - file system consistency check and interactive repair

SYNTAX

fsck [option] ... [filesystem] ...

DESCRIPTION

Fsck audits and interactively repairs inconsistent conditions for the named filesystems. Fsck ignores the 'file system clean' flag in the super block; upon completion fsck sets 'file system clean' (if it was not already set).

If a file system is consistent then the number of files, number of blocks used, and number of blocks free are reported. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. Most corrections lose data; all losses are reported. The default action for each correction is to wait for the operator to respond 'yes' or 'no'. Without write permission fsck defaults to -n action.

These options are recognized:

-y Assume a yes response to all questions.

-n Assume a no response to all questions.

-sX Ignore the actual free list and (unconditionally) construct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done, or extreme care should be taken that the system is quiescent and that it is rebooted immediately afterwards. This precaution is necessary so that the old, bad, in-core copy of the superblock will not continue to be used, or written on the file system.

The free list is created with interleaving according to the following specification of c:s for X:
-sc:s space free blocks s blocks apart in cylinders of c blocks each.

If X is not given, the values used when the filesystem was created are used. If these values were not specified, then c=400, s=9 is assumed.

-SX Conditionally reconstruct the free list. This option is like -sX except that the free list is rebuilt only if there were no discrepancies discovered in the file system. It is useful for forcing free list reorganization on uncontaminated file systems. -S forces -n.

-t If fsck cannot obtain enough memory to keep its tables,

it uses a scratch file. If the -t option is specified, the file named in the next argument is used as the scratch file. Without the -t option, fsck prompts if it needs a scratch file. The file should not be on the file system being checked, and if it is not a special file or did not already exist, it is removed when fsck completes.

If no filesystems are given to fsck then a default list of file systems is read from the file **/etc/checklist**.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.

2. Blocks claimed by an inode or the free list outside the range of the file system.

3. Incorrect link counts.

4. Size checks:
   Incorrect number of blocks in file.
   Directory size not a multiple of 16 bytes.

5. Bad inode format.

6. Blocks not accounted for anywhere.

7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.

8. Super Block checks:
   More than 65536 inodes.
   More blocks for inodes than there are in the file system.

9. Bad free block list format.

10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the "lost+found" directory. The name assigned is the inode number. The only restriction is that the directory "lost+found" must preexist in the root of the filesystem being checked and must have empty slots in which entries can be made. This is accomplished by making "lost+found", copying a number of files to the directory, and then removing them (before fsck is executed).

Checking the raw device is almost always faster.

FILES

/etc/checklist default list of file systems to check.
lost+found home for orphans

SEE ALSO

dcheck(1), icheck(1), filsys(5), crash(8), mount(1M)
/etc/rc the system startup script which uses fsck heavily.

NOTES

Inode numbers for . and .. in each directory should be checked for validity.
The -b option of icheck(1) should be available.

NAME

gets - get a string from standard input

SYNTAX

gets [default]

DESCRIPTION

Gets can be used with csh(1S) to read a string from the standard input. If a default is given it is used if an error occurs. The resulting string (either the default or as read from the standard input is written to the standard output. If no default is given and an error occurs, gets exits with exit status 1.

SEE ALSO

csh(1S)

NAME

graph - draw a graph

SYNTAX

graph [option] ...

DESCRIPTION

Graph with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by the plot(1) filters.

If the coordinates of a point are followed by a nonnumeric string, that string is printed as a label beginning on the point. Labels may be surrounded with quotes "...", in which case they may be empty or contain blanks and numbers; labels never contain newlines.

The following options are recognized, each as a separate argument.

-a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument (default 1). A second optional argument is the starting point for automatic abscissas (default 0 or lower limit given by **-x**).

-b Break (disconnect) the graph after each label in the input.

-c Character string given by next argument is default label for each point.

-g Next argument is grid style, 0 no grid, 1 frame with ticks, 2 full grid (default).

-l Next argument is label for graph.

-m Next argument is mode (style) of connecting lines: 0 disconnected, 1 connected (default). Some devices give distinguishable line styles for other small integers.

-s Save screen, don't erase before plotting.

-x [l]
If l is present, x axis is logarithmic. Next l (or 2) arguments are lower (and upper) x limits. Third argument, if present, is grid spacing on x axis. Normally these quantities are determined automatically.

-y [l]

Similarly for y.

-h Next argument is fraction of space for height.

-w Similarly for width.

-r Next argument is fraction of space to move right before plotting.

-u Similarly to move up before plotting.

-t Transpose horizontal and vertical axes. (Option -x now applies to the vertical axis.)

A legend indicating grid range is produced with a grid unless the -s option is present.

If a specified lower limit exceeds the upper limit, the axis is reversed.

SEE ALSO

spline(1), plot(1)

NOTES

Graph stores all points internally and drops those for which there isn't room.
Segments that run out of bounds are dropped, not windowed.
Logarithmic axes may not be reversed.

NAME

grep - search a file for a limited regular expression

SYNTAX

grep [option] ... expression [file] ...

DESCRIPTION

Commands of the grep family search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The file name is shown if there is more than one input file.

Grep patterns are limited regular expressions in the style of ed(1). Use egrep for searches of full regular expressions and fgrep for fast searches of fixed strings.

The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its linenumber in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s No output is produced, but the correct exit status is returned. (See DIAGNOSTICS below.) Useful for detecting a given pattern in a file before performing an operation on it.

-h Do not print filename headers with output lines.

-y Alphabetic letters in the pattern will match letters of either case in the input.

-e expression
Same as a simple expression argument, but useful when the expression begins with a -.

Care should be taken when using the following characters in an expression as they are also meaningful to the shell: $ * [^ | ? ' " ( ) and \. It is safest to enclose the entire expression argument in single quotes ' '.

Regular expressions are composed of the following elements. Note that in the following description a 'character'

excludes the newline character:

A backslash (\) followed by a single character matches that character. This is true for all of the characters discussed below, so that the special meaning of any character ceases if preceded by a backslash.

A caret (^). matches the beginning of a line.

A dollar sign ($) matches the end of a line.

A period (.) matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in "a-z0-9". An end bracker (]) may occur only as the first character of the string. A literal hyphen (-) must be placed where it can't be mistaken as a range indicator.

Parentheses escaped with backslashes \(and \) can be used to delimit and remember expressions. Remembered expressions are numbered beginning with 1 in the order in which they are encountered. Each delimited expression can then be repeated in the search expression by escaping the appropriate expression number. For example, \(TO\)\1 would match "TOTO" and \(T\)\(O\)\2\1 would match "TOOT"

EXAMPLES

All examples below find regular expressions in the files names file1 and file2 .

Find all lines beginning with an upper case letter:

```
grep '^[A-Z]' file1 file2
```

Find all lines containing an asterisk:

```
grep '\*' file1 file2
```

Find all lines ending with a backslash:

```
grep '\\$' file1 file2
```

Find all lines not beginning with ".H":

```
grep -v '^\.H' file1 file2
```

Find all lines containing exactly three characters:

```
egrep '...' file1 file2
```

SEE ALSO

ed(1), egrep(1), fgrep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

NOTES

Ideally there should be only one grep, but no known single algorithm exists that spans a wide enough range of space-time tradeoffs.

Lines are limited to 256 characters; longer lines are truncated.

NAME

haltsys - close out the file systems and halt the CPU

SYNTAX

/etc/haltsys

DESCRIPTION

Haltsys does a shutdn() system call (see shutdn(2)) to flush out pending disk I/O, mark the file systems clean, and halt the processor. Haltsys takes effect immediately, so user processes should be killed beforehand. Shutdown(1M) is recommended for normal system termination; it warns the users, cleans things up, and calls haltsys. Use haltsys directly only if some system problem prevents the running of shutdown.

SEE ALSO

shutdn(2), shutdown(1M), boot(8)

NAME

head - print first few lines of a stream

SYNTAX

head [-count] [file ...]

DESCRIPTION

This filter prints the first count lines of each of the specified files. If no files are specified, head reads from the standard input. If no count is specified, then 10 lines are printed.

SEE ALSO

tail(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NAME

icheck - file system storage consistency check

SYNTAX

icheck [-s] [-b numbers] [filesystem]

DESCRIPTION

Icheck examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. If the file system is not specified, a set of default file systems is checked. The normal output of icheck includes a report of

The total number of files and the numbers of regular, directory, block special and character special files.

The total number of blocks in use and the numbers of single-, double-, and triple-indirect blocks and directory blocks.

The number of free blocks.

The number of blocks missing; i.e. not in any file nor in the free list.

The -s option causes icheck to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The -s option causes the normal output reports to be suppressed.

Following the -b option is a list of block numbers; whenever any of the named blocks turns up in a file, a diagnostic is produced.

Icheck is faster if the raw version of the special file is used, since it reads the i-list many blocks at a time.

FILES

Default file systems vary with installation.

SEE ALSO

dcheck(1), ncheck(1), filsys(5), clri(1)

DIAGNOSTICS

For duplicate blocks and bad blocks (which lie outside the file system) icheck announces the difficulty, the i-number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and icheck considers it to contain 0. 'Bad freeblock' means that a block number outside the available space was encountered in the free list. 'n dups in free' means that n blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

NOTES

Since icheck is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super-blocks and consequently can get core images.

NAME

join - relational database operator

SYNTAX

join [options] filel file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of filel and file2. If filel is '-', the standard input is used.

Filel and file2 must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in filel and file2 that have identical join fields. The output line normally consists of the common field, then the rest of the line from filel, then the rest of the line from file2.

Fields are normally separated by blank, tab or newline. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

-an In addition to the normal output, produce a line for each unpairable line in file n, where n is 1 or 2.

-e s Replace empty output fields by string s.

-jn m
Join on the mth field of file n. If n is missing, use the mth field in each file.

-o list
Each output line comprises the fields specifed in list, each element of which has the form n.m, where n is a file number and m is a field number.

-tc Use character c as a separator (tab character). Every appearance of c in a line is significant.

SEE ALSO

sort(1), comm(1), awk(1)

NOTES

With default field separation, the collating sequence is that of sort -b; with -t, the sequence is that of a plain sort.

The conventions of join, sort, comm, uniq, look and awk(1) are wildly incongruous.

NAME

kill - terminate a process with extreme prejudice

SYNTAX

kill [-signo] processid ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. If a signal number preceded by '-' is given as first argument, that signal is sent instead of terminate (see signal(2)). This will kill processes that do not catch the signal; in particular 'kill -9 ...' is a sure kill.

By convention, if process number 0 is specified, all members in the process group (i.e. processes resulting from the current login) are signaled.

The killed processes must belong to the current user unless he is the super-user. To shut the system down and bring it up single user the super-user may use 'kill -1 1'; see init(8).

The process number of an asynchronous process started with '&' is reported by the shell. Process numbers can also be found by using ps(1).

SEE ALSO

ps(1), kill(2), signal(2)

NAME

l - list information about contents of directory

SYNTAX

l [-asdrucifg] name ...

DESCRIPTION

For each directory argument, l lists the contents of the directory; for each file argument, l repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents. By default, information is listed in the format of the 'ls -l' command, which is identical to l in most respects. This format provides mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers. There are several options:

-t Sort by time modified (latest first) instead of by name, as is normal.

-a List all entries; usually '.' and '..' are suppressed.

-s Give size in blocks, including indirect blocks, for each entry.

-d If argument is a directory, list only its name, not its contents (mostly used with -l to get status on directory).

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-u Use time of last access instead of last modification for sorting (-t) or printing (-l).

-c Use time of last modification to inode (mode, etc.) instead of last modification to file for sorting (-t) or printing (-l).

-i Print i-number in first column of the report for each file listed.

-f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -l, -t, -s, and -r, and turns on -a; the order is the order in which entries appear in the directory.

-g Give group ID instead of owner ID in long listing.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as s if the file has set-group-ID mode; likewise the user-execute permission character is given as s if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '-') is t if the 1000 bit of the mode is on. See chmod(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for 'l'.
/etc/group to get group ID's for 'l -g'.

NAME

lc - list directory in columns

SYNTAX

lc [-abcdfgilmqrstuxlCFR] name ...
l [lc options] name ...

DESCRIPTION

For each directory argument, lc lists the contents of the directory; for each file argument, lc repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The format chosen depends on whether the output is going to a teletype, and may also be controlled by option flags. The default format for a teletype is to list the contents of directories in multi-column format, with the entries sorted down the columns. (Files which are not the contents of a directory being interpreted are always sorted across the page rather than down the page in columns. This is because the individual file names may be arbitrarily long.) If the standard output is not a teletype, the default format is to list one entry per line. Finally, there is a stream output format in which files are listed across the page, separated by ',' characters. The -m flag enables this format; when invoked as l this format is also used.

There are an unbelievable number of options:

-l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

-t Sort by time modified (latest first) instead of by name, as is normal.

-a List all entries; usually '.' and '..' are suppressed.

-s Give size in blocks, including indirect blocks, for each entry.

-d If argument is a directory, list only its name, not its contents (mostly used with -l to get status on directory).

-r Reverse the order of sort to get reverse alphabetic or

oldest first as appropriate.

-u Use time of last access instead of last modification for sorting (-t) or printing (-l).

-c Use time of file creation for sorting or printing.

-i Print i-number in first column of the report for each file listed.

-f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -l, -t, -s, and -r, and turns on -a; the order is the order in which entries appear in the directory.

-g Give group ID instead of owner ID in long listing.

-m force stream output format

-1 force one entry per line output format, e.g. to a teletype

-C force multi-column output, e.g. to a file or a pipe

-q force printing of non-graphic characters in file names as the character '?'; this normally happens only if the output device is a teletype

-b force printing of non-graphic characters to be in the \ddd notation, in octal.

-x force columnar printing to be sorted across rather than down the page; this is the default if the last character of the name the program is invoked with is an 'x'.

-F cause directories to be marked with a trailing '/' and executable files to be marked with a trailing '*'; this is the default if the last character of the name the program is invoked with is a 'f'.

-R recursively list subdirectories encountered.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
m if the entry is a multiplexor-type character special file;
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as s if the file has set-group-ID mode; likewise the user-execute permission character is given as s if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '-') is t if the 1000 bit of the mode is on. See chmod(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for 'lc -l'.
/etc/group to get group ID's for 'lc -g'.

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "lc -s" is much different than "lc -s | lpr". On the other hand, not doing this would be incompatible with older shell scripts using ls, as is expected that many will replace the older ls with lc by aliasing lc to ls.

Column widths choices are poor for terminals which can tab.

NAME

ld - link editor

SYNTAX

ld [option] file ...

DESCRIPTION

Ld combines several object programs into one, resolves external references, and searches libraries. Ld combines the given object files, producing an object module which can be either executed or become the input for a further ld run. (In the latter case, the -r option must be given to preserve the relocation records.) The output of ld is left by default in the file a.out. This file is made executable only if no errors occurred.

The files given as arguments are concatenated in the order specified. The default entry point of the output is the beginning of the first routine in the first file. The C compiler, cc , calls ld automatically unless given the **-c** option. The command line that cc passes to ld is

ld /lib/crt0.o files cc-options -lc

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by ranlib(1S), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. If the first member of a library is named '__.SYMDEF', then it is understood to be a dictionary for the library such as produced by ranlib; the dictionary is searched iteratively to satisfy as many references as possible.

The symbols '_etext', '_edata' and '_end' ('etext', 'edata' and 'end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data, respectively. It is erroneous to define these symbols.

If no errors occur and there are no unresolved external references, then short form relocation information is attached and the file is made executable. This short form relocation information is sufficient to allow the file to be used for another pass of ld , to change the text and data base addresses. At the same time, the **-n** , **-i** , or **-F** options can be used to produce different types of executable files.

Ld understands several options. Except for -l, they should appear before the names of all object file arguments.

-s 'Strip' the output to save space by removing the symbol table and relocation records. Note that stripping impairs the usefulness of the debugger. This information can also be removed later with strip(1S).

-sr Do not attach the short form of relocation. This does not imply removing the symbol table, as with -s .

-u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-lx This option is an abbreviation for the library name '/lib/libx.a', where x is a string. If the library does not exist, ld then tries '/usr/lib/libx.a'. A library is searched when its name is encountered, so the placement of a -l is significant. Note that -l with no argument, defaults to -lc . If the processor on which ld is running is not the same as the target processor, then it is possible that -p may be implied. In the case of the MC68000 target, -p /usr/lib/m68lib is implied.

-p Take the following argument as the directory in which -lx libraries will be found.

-x Do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

-X Save local symbols except for those whose names begin with 'L'. This option is used by cc(1S) to discard internally generated labels while retaining symbols local to routines.

-r Generate (long form) relocation records in the output file so that the output file can be the subject of another ld run. This flag also prevents final definitions from being given to common symbols and suppresses the 'undefined symbol' diagnostics.

-d Force definition of common storage even if the -r flag is present.

-nn Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data

areas up to the first possible page boundary following the end of the text. A warning is issued if the current machine does not support this option.

-nr Identical to **-nn** except that the text and data positions are reversed.

-n Identical to whichever of **-nn** and **-nr** is the default for the current machine.

-i When the output file is executed, the program text and data areas are given separate address spaces. The only difference between this option and **-n** is that with -i the data may start at a boundary unrelated to the position of the text. A warning is issued if the current machine does not support this option.

-o The name argument after -o is used as the name of the ld output file, instead of a.out.

-e The following argument is taken to be the name of the entry point of the loaded program. The base of the text segment is the default.

-D The next argument is a decimal number that sets the size of the data segment.

-N The next argument is taken to be a hexidecimal number that sets the pagesize, or rounding size, for use with the **-n** option. With -i, it specifies the base of the data segment. With **-nn**, it is used to compute the base of the data segment. With **-nr**, it is used to compute the base of the text segment.

-R The next argument is taken to be a hexidecimal number that is used as the base address for text relocation. With -i or **-nn** , it also specifies the text base address; with **-nr** it specifies the data base address.

-F The next argumen is taken to be a hexadecimal number that specifies the size of the stack required by the object file when executing. This only has meaning on those processors that cannot expand the stack dynamically.

FILES

/lib/lib*.a	libraries
/usr/lib/lib*.a	more libraries
a.out	output file

SEE ALSO

as(1S), ar(1S), cc(1S), ranlib(1S), strip(1S), a.out(5)

NAME

learn - computer aided instruction about XENIX

SYNTAX

learn [**- directory**] [subject [lesson [speed]]]

DESCRIPTION

This command gives computer aided courses and practice in XENIX. To get started, type 'learn'. The program will ask questions to find out what you want to do. The questions may be bypassed by naming a subject, and the last lesson number that learn told you in the previous session. You may also include a speed number that was given with the lesson number (but without the parentheses that learn places around the speed number). If lesson is '-', learn prompts for each lesson; this is useful when debugging.

Among the subjects handled are:

Files
More Files
Ed
C
Macros
Eqn

The special command 'bye' terminates a learn session.

The -directory option allows use of learn in a non-standard place.

FILES

/usr/lib/learn and all dependent directories and files

SEE ALSO

man(1)

NOTES

The main strength of learn, that it asks the student to use the real XENIX, also makes possible baffling mistakes. It is helpful, especially for nonprogrammers, to have a XENIX initiate near by during the first sessions.

Occasionally, lessons are incorrect because the local version of a command works in a non-standard way.

NAME

lex - generator of lexical analysis programs

SYNTAX

lex [**-tvfn**] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analyis of text. The input files (standard input default) contain regular expressions to be searched for, and actions written in C to be executed when expressions are found.

A C source program, 'lex.yy.c' is generated, to be compiled thus:

```
cc lex.yy.c -lln
```

This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The following lex program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%%
[A-Z] putchar(yytext[0]+'a'-'A');
[ ]+$
[ ]+  putchar(' ');
```

The options have the following meanings.

-t Place the result on the standard output instead of in file 'lex.yy.c'.

-v Print a one-line summary of statistics of the generated analyzer.

-n Opposite of **-v**; **-n** is default.

-f 'Faster' compilation: don't bother to pack the resulting tables; limited to small programs.

SEE ALSO

yacc(1S)
M. E. Lesk and E. Schmidt, LEX - Lexical Analyzer Generator

NAME

lint - a C program verifier

SYNTAX

lint [**-abchnpuvx**] file ...

DESCRIPTION

Lint attempts to detect features of the C program files which are likely to be bugs, or non-portable, or wasteful. It also checks the type usage of the program more strictly than the compilers. Among the things which are currently found are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

By default, it is assumed that all the files are to be loaded together; they are checked for mutual compatibility. Function definitions for certain libraries are available to lint; these libraries are referred to by a conventional name, such as '-lm', in the style of ld(1S).

Any number of the options in the following list may be used. The **-D**, **-U**, and **-I** options of cc(1S) are also recognized as separate arguments.

p Attempt to check portability to the IBM and GCOS dialects of C.

h Apply a number of heuristic tests to attempt to intuit bugs, improve style, and reduce waste.

b Report break statements that cannot be reached. (This is not the default because, unfortunately, most lex and many yacc outputs produce dozens of such comments.)

v Suppress complaints about unused arguments in functions.

x Report variables referred to by extern declarations, but never used.

a Report assignments of long values to int variables.

c Complain about casts which have questionable portability.

u Do not complain about functions and variables used and not defined, or defined and not used (this is suitable

for running lint on a subset of files out of a larger program).

n Do not check compatibility against the standard library.

Exit(2) and other functions which do not return are not understood; this causes various lies.

Certain conventional comments in the C source will change the behavior of lint:

/*NOTREACHED*/
at appropriate points stops comments about unreachable code.

/*VARARGSn*/
suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first n arguments are checked; a missing n is taken to be 0.

/*NOSTRICT*/
shuts off strict type checking in the next expression.

/*ARGSUSED*/
turns on the **-v** option for the next function.

/*LINTLIBRARY*/
at the beginning of a file shuts off complaints about unused functions in this file.

FILES

/usr/lib/lint[12] programs
/usr/lib/llib-lc declarations for standard functions
/usr/lib/llib-port declarations for portable functions

SEE ALSO

cc(1S)
S. C. Johnson, Lint, a C Program Checker

NAME

ln - make a link

SYNTAX

ln name1 [name2]

DESCRIPTION

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, etc.) may have several links to it. There is no way to distinguish a link to a file from its original directory entry; any changes in the file are effective independently of the name by which the file is known.

Ln creates a link to an existing file name1. If name2 is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of name1.

It is forbidden to link to a directory or to link across file systems.

SEE ALSO

rm(1)

NAME

login - sign on

SYNTAX

login [username]

DESCRIPTION

The login command is used when a user initially signs on, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. See 'How to Get Started' for how to dial up initially.

If login is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it will not appear on the written record of the session.

After a successful login, accounting files are updated and the user is informed of the existence of mail and message-of-the-day files. Login initializes the user and group IDs and the working directory, then executes a command interpreter (usually sh(1)) according to specifications found in a password file. Argument 0 of the command interpreter is '-sh.

Login is recognized by sh(1) and executed directly (without forking).

FILES

/etc/utmp	accounting
/usr/adm/wtmp	accounting
/usr/spool/mail/*mail	
/etc/motd	message-of-the-day
/etc/passwd	password file

SEE ALSO

init(8), newgrp(1), getty(8), mail(1), passwd(1), passwd(5)

DIAGNOSTICS

'Login incorrect,' if the name or the password is bad.
'No Shell', 'cannot open password file', 'no directory': consult a programming counselor.

NAME

look - find lines in a sorted list

SYNTAX

look [**-df**] string [file]

DESCRIPTION

Look consults a sorted file and prints all lines that begin with string. It uses binary search.

The options **d** and **f** affect comparisons as in sort(1):

d 'Dictionary' order: only letters, digits, tabs and blanks participate in comparisons.

f Fold. Upper case letters compare equal to lower case.

If no file is specified, /usr/dict/words is assumed with collating sequence **-df.**

FILES

/usr/dict/words

SEE ALSO

sort(1), grep(1)

NAME

lorder - find ordering relation for an object library

SYNTAX

lorder file ...

DESCRIPTION

The input is one or more object or library archive (see ar(1S)) files. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by tsort(1S) to find an ordering of a library suitable for one-pass access by ld(1S).

This brash one-liner intends to build a new library from existing '.o' files.

```
ar cr library `lorder *.o | tsort`
```

FILES

*symref, *symdef
nm(1S), sed(1), sort(1), join(1)

SEE ALSO

tsort(1S), ld(1S), ar(1S)

NOTES

The names of object files, in and out of libraries, must end with '.o'; nonsense results otherwise.

NAME

lpr, vpr - line printer spooler

SYNTAX

lpr [option] ... [file] ...
vpr [**-b** banner] [file]

DESCRIPTION

Lpr causes the files to be queued for printing on a line printer. If no files are named, the standard input is read. The following options are available:

-r Remove the file when it has been queued.

-c Copy the file to insulate against changes that may happen before printing.

-m Report by mail(1) when printing is complete.

-n Do not report by mail. This is the default option.

Vpr is the program used by lpr when the online printer is a Versatec machine to insert an identification banner before the output, strip overstrikes, and, where possible, remove blank lines to make 66-line pages fit on 64 lines. If the file /usr/adm/vpacct is writable, vpr places accounting information on it.

FILES

/usr/spool/lpd/lock
/usr/spool/lpd/cf* data file
/usr/spool/lpd/df* daemon control file
/usr/spool/lpd/tf* temporary version of control file
/usr/bin/vpr for Versatec printer
/usr/adm/vpacct

SEE ALSO

lpd(8)

NAME

ls - list contents of directory

SYNTAX

ls [**-ltasdrucifg**] name ...

DESCRIPTION

For each directory argument, ls lists the contents of the directory; for each file argument, ls repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents. There are several options:

-l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

-t Sort by time modified (latest first) instead of by name, as is normal.

-a List all entries; usually '.' and '..' are suppressed.

-s Give size in blocks, including indirect blocks, for each entry.

-d If argument is a directory, list only its name, not its contents (mostly used with -l to get status on directory).

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-u Use time of last access instead of last modification for sorting (**-t**) or printing (-l).

-c Use time of last modification to inode (mode, etc.) instead of last modification to file for sorting (**-t**) or printing (-l).

-i Print i-number in first column of the report for each file listed.

-f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -l, **-t**, -s, and **-r**, and turns on **-a**; the order is the order in which entries appear in the directory.

-g Give group ID instead of owner ID in long listing.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise the user-execute permission character is given as **s** if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '-') is **t** if the 1000 bit of the mode is on. See chmod(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for 'ls -l'.
/etc/group to get group ID's for 'ls -g'.

NAME

m4 - macro processor

SYNTAX

m4 [files]

DESCRIPTION

M4 is a macro processor intended as a front end for C and other languages. Each of the argument files is processed in order; if there are no arguments, or if an argument is '-', the standard input is read. The processed text is written on the standard output.

Macro calls have the form

name(arg1,arg2, . . . , argn)

The '(' must immediately follow the name of the macro. If a defined macro name is not followed by a '(', it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Potential macro names consist of alphabetic letters, digits, and underscore '_', where the first character is not a digit.

Left and right single quotes (`') are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $n in the replacement text, where n is a digit, is replaced by the n-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string.

undefine removes the definition of the macro named in its argument.

ifdef If the first argument is defined, the value is the

second argument, otherwise the third. If there is no third argument, the value is null. The word unix is predefined on UNIX versions of m4.

changequote
Change quote characters to the first and second arguments. Changequote without arguments restores the original values (i.e., `').

divert M4 maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The divert macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum returns the value of the current output stream.

dnl reads and discards characters up to and including the next newline.

ifelse has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

eval evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, -, *, /, %, ^ (exponentiation); relationals; parentheses.

len returns the number of characters in its argument.

index returns the position in its first argument where the second argument begins (zero origin), or -1 if the second argument does not occur.

substr returns a substring of its first argument. The

second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include returns the contents of the file named in the argument.

sinclude is identical to include, except that it says nothing if the file is inaccessible.

syscmd executes the XENIX command given in the first argument. No value is returned.

maketemp fills in a string of XXXXX in its argument with the current process id.

errprint prints its argument on the diagnostic output file.

dumpdef prints current names and definitions, for the named items, or for all if no arguments are given.

SEE ALSO

B. W. Kernighan and D. M. Ritchie, The M4 Macro Processor

DIAGNOSTICS

pushback overflow: an expansion caused overflow of m4's internal buffers. Usually the result of a looping recursive definition.

NAME

mail - send or receive mail among users

SYNTAX

mail person ...
mail [-r] [-q] [-p] [**-f** file]

DESCRIPTION

Mail with no argument prints a user's mail, message-by-message, in last-in, first-out order; the optional argument -r causes first-in, first-out order. If the -p flag is given, the mail is printed with no questions asked; otherwise, for each message, mail reads a line from the standard input to direct disposition of the message.

newline
Go on to next message.

d Delete message and go on to the next.

p Print message again.

- Go back to previous message.

s [file] ...
Save the message in the named files ('mbox' default).

w [file] ...
Save the message, without a header, in the named files ('mbox' default).

m [person] ...
Mail the message to the named persons (yourself is default).

EOT (control-D)
Put unexamined mail back in the mailbox and stop.

q Same as EOT.

x Exit, without changing the mailbox file.

!command
Escape to the Shell to do command.

? Print a command summary.

An interrupt stops the printing of the current letter. The optional argument -q causes mail to exit after interrupts without changing the mailbox.

When persons are named, mail takes the standard input up to an end-of-file (or a line with just '.') and adds it to each person's 'mail' file. The message is preceded by the sender's name and a postmark. Lines that look like postmarks are prepended with '>'. A person is usually a user name recognized by login(1). To denote a recipient on a remote system, prefix person by the system name and exclamation mark (see uucp(1)).

The **-f** option causes the named file, e.g. 'mbox', to be printed as if it were the mail file.

Each user owns his own mailbox, which is by default generally readable but not writable. The command does not delete an empty mailbox nor change its mode, so a user may make it unreadable if desired.

When a user logs in he is informed of the presence of mail.

FILES

/usr/spool/mail/* mailboxes
/etc/passwd to identify sender and locate persons
mbox saved mail
/tmp/ma* temp file
dead.letter unmailable text
uux(1)

SEE ALSO

write(1), uucp(1)

NOTES

There is a locking mechanism intended to prevent two senders from accessing the same mailbox, but it is not perfect and races are possible.

NAME

make - maintain program groups

SYNTAX

make [-f makefile] [option] ... file ...

DESCRIPTION

Make executes commands in makefile to update one or more target names. Name is typically a program. If no -f option is present, 'makefile' and 'Makefile' are tried in order. If makefile is '-', the standard input is taken. More than one -f option may appear

Make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon, and all following lines that begin with a tab, are shell commands to be executed to update the target.

Sharp and newline surround comments.

The following makefile says that 'pgm' depends on two files 'a.o' and 'b.o', and that they in turn depend on '.c' files and a common file 'incl'.

```
pgm: a.o b.o
     cc a.o b.o -lm -o pgm
a.o: incl a.c
     cc -c a.c
b.o: incl b.c
     cc -c b.c
```

Makefile entries of the form

```
string1 = string2
```

are macro definitions. Subsequent appearances of $(string1) are replaced by string2. If string1 is a single character, the parentheses are optional.

Make infers prerequisites for files for which makefile gives no construction commands. For example, a '.c' file may be inferred as prerequisite for a '.o' file and be compiled to produce the '.o' file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
     cc a.o b.o -lm -o pgm
a.o b.o: incl
```

Prerequisites are inferred according to selected suffixes listed as the 'prerequisites' for the special name '.SUFFIXES'; multiple lists accumulate; an empty list clears what came before. Order is significant; the first possible name for which both a file and a rule as described in the next paragraph exist is inferred. The default list is

```
.SUFFIXES: .out .o .c .e .r .f .y .l .s
```

The rule to create a file with suffix s2 that depends on a similarly named file with suffix s1 is specified as an entry for the 'target' s1s2. In such an entry, the special macro $* stands for the target name with suffix deleted, $@ for the full target name, $< for the complete list of prerequisites, and $? for the list of prerequisites that are out of date. For example, a rule for making optimized '.o' files from '.c' files is

```
.c.o: ; cc -c -O -o $@ $*.c
```

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, 'CFLAGS' is used for cc and f77(1S) options, 'LFLAGS' and 'YFLAGS' for lex and yacc(1S) options.

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the special target '.SILENT' is in makefile, or the first character of the command is '@'.

Commands returning nonzero status (see intro(1)) cause make to terminate unless the special target '.IGNORE' is in makefile or the command begins with <tab><hyphen>.

Interrupt and quit cause the target to be deleted unless the target depends on the special name '.PRECIOUS'.

Other options:

-i Equivalent to the special entry '.IGNORE:'.

-k When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.

-n Trace and print, but do not execute the commands needed to update the targets.

-t Touch, i.e. update the modified date of targets, without executing any commands.

-r Equivalent to an initial special entry '.SUFFIXES:' with no list.

-s Equivalent to the special entry '.SILENT:'.

FILES

makefile, Makefile

SEE ALSO

sh(1), touch(1)
S. I. Feldman Make - A Program for Maintaining Computer Programs

NOTES

Some commands return nonzero status inappropriately. Use -i to overcome the difficulty.
Commands that are directly executed by the shell, notably cd(1), are ineffectual across newlines in make.

NAME

man - print sections of this manual

SYNTAX

man [option ...] [chapter] title ...

DESCRIPTION

Man locates and prints the section of this manual named title in the specified chapter. (In this context, the word 'page' is often used as a synonym for 'section'.) The title is entered in lower case. The chapter number does not need a letter suffix. If no chapter is specified, the whole manual is searched for title and all occurrences of it are printed.

Options and their meanings are:

-t Phototypeset the section using troff(1T).

-n Print the section on the standard output using nroff(1T).

-e Appended or prefixed to any of the above causes the manual section to be preprocessed by neqn or eqn(1T); **-e** alone means **-te**.

-w Print the path names of the manual sections, but do not print the sections themselves.

(default)
Copy an already formatted manual section to the terminal, or, if none is available, act as **-n**. It may be necessary to use a filter to adapt the output to the particular terminal's characteristics.

Further options, e.g. to specify the kind of terminal you have, are passed on to troff(1T) or nroff(1T). Options and chapter may be changed before each title.

For example:

man man

would reproduce this section, as well as any other sections named man that may exist in other chapters of the manual, e.g. man(7).

FILES

/usr/man/man?/*
/usr/man/cat?/*

SEE ALSO

nroff(1T), eqn(1T), man(7T)

NOTES

The manual is supposed to be reproducible either on a phototypesetter or on a terminal. However, on a terminal some information is necessarily lost.

NAME

mesg - permit or deny messages

SYNTAX

mesg [n] [y]

DESCRIPTION

Mesg with argument n forbids messages via write(1) by revoking non-user write permission on the user's terminal. Mesg with argument y reinstates permission. All by itself, mesg reports the current state without changing it.

FILES

/dev/tty*
/dev

SEE ALSO

write(1)

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

mkdir - make a directory

SYNTAX

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777. Standard entries, '.', for the directory itself, and '..' for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

rm(1)

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made. Otherwise it prints a diagnostic and returns nonzero.

NAME

mkfs - construct a file system

SYNTAX

/etc/mkfs [-y] [-n] special proto **[m n]**

DESCRIPTION

Mkfs constructs a file system by writing on the special file special according to the directions found in the prototype file proto. The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program. The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of i-nodes in the i-list. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **-bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or - to specify set-user-id mode or not. The third is g or - for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see chmod(1).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, mkfs makes the entries . and .. and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token $.

If the prototype file cannot be opened and its name consists of a string of digits, mkfs builds a file system with a single empty directory on it. The size of the file system is the value of proto interpreted as a decimal number. The number of i-nodes is calculated as a function of the filsystem size. The boot program is left uninitialized.

Mkfs can also be used to create a file system image in a regular file, rather than on a special device file, by giving the pathname of the target file, instead of special.

If the target file is not a regular file, then mkfs checks for an existing file system on that device. If it appears the device contains a file system, operator confirmation is requested before overwriting the data. The -y switch overrides this, and will write over any existing data without question. The -n switch causes mkfs to terminate without question if the target contains an existing file system. The check used is to read block one from the target device (block one is the super block) and see whether the bytes are all the same. If they are not, this is taken to be meaningful data, and confirmation is requested.

A sample prototype specification follows:

```
/usr/mdec/uboot
4872 55
d--777 3 1
usr  d--777 3 1
     sh   ---755 3 1 /bin/sh
     ken  d--755 6 1
          $
     b0   b--644 3 1 0 0
     c0   c--644 3 1 0 0
     $
$
```

SEE ALSO

filsys(5), dir(5),

NOTES

There should be some way to specify links.

NAME

mknod - build special file

SYNTAX

/etc/mknod name [**c**] [**b**] major minor

DESCRIPTION

Mknod makes a special file. The first argument is the name of the entry. The second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the major device type and the minor device (e.g. unit, drive, or line number).

The assignment of major device numbers is specific to each system. They can be determined by examining the system source file c.c.

SEE ALSO

mknod(2)

NAME

mkstr - create an error message file by massaging C source

SYNTAX

mkstr [-] messagefile prefix file ...

DESCRIPTION

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

Mkstr will process each of the specified files, placing a massaged version of the input file in a file whose name consists of the specified prefix and the original name. A typical usage of mkstr would be

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file pistrings and processed copies of the source for these files to be placed in files whose names are prefixed with xx.

To process the error messages in the source to the message file mkstr keys on the string 'error("' in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a new-line character; the null character terminates the message so it can be easily used when retrieved, the new-line character makes it possible to sensibly cat the error message file to see its contents. The massaged copy of the input file then contains a lseek pointer into the file which can be used to retrieve the message, i.e.:

```
char efilname[] =  "/usr/lib/pi_strings";
int  efil = -1;

error(a1, a2, a3, a4)
{
     char buf[256];

     if (efil < 0) {
          efil = open(efilname, 0);
          if (efil < 0) {
oops:
               perror(efilname);
               exit(1);
          }
```

```
        }
        if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
            goto oops;
        printf(buf, a2, a3, a4);
    }
```

The optional - causes the error messages to be placed at the end of the specified message file for recompiling part of a large mkstred program.

SEE ALSO

lseek(2), xstr(1S)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

All the arguments except the name of the file to be processed are unnecessary.

NAME

mkuser - add a login ID to the system

SYNTAX

/etc/mkuser

DESCRIPTION

Mkuser is used to add more user login ID's to the system. It is the prefered method for adding new users to the system, since it handles all directory creation and password file update.

To add a new user to the system, mkuser requires three pieces of information: the login name, the initial password, and an optional comment string for the password file. The program prompts for these three items and validates the given data. The login name is checked against certain criteria (must be at least three characters, and begin with a lowercase letter). The password must follow standard XENIX conventions, see passwd(1). The password file comment field can be up to 20 characters of information.

Mkuser takes some of its parameters from a default file, /etc/default/mkuser. Currently the two settable options are the pathname for the login shell and the root path of home directories. An example default file is:

```
home=/usr
shell=/bin/sh
```

This file can be edited (by the super-user) to change these defaults. There are three other files in the directory /usr/lib which may also be altered to suit local options. They are mkuser.help which is the introductory explanation given by mkuser on startup, mkuser.mail which is the initial mail message sent to new users, and mkuser.prof, the standard .profile file given to new users.

Mkuser allocates user ID's starting at 200, or the largest number used in the password file. It allocates all new users to the initial group 50.

Mkuser can only be executed by the super-user.

FILES

/etc/passwd
/usr/spool/mail/username.
/etc/default/mkuser
/usr/lib/mkuser.help
/usr/lib/mkuser.prof
/usr/lib/mkuser.mail

SEE ALSO

rmuser(1m), passwd(1).

NAME

more, page - view file one screenful at a time

SYNTAX

more [-cdflsur] [-n] [+linenumber] [+/pattern] [name ...]

page more options

DESCRIPTION

This filter allows examination of a continuous text one screenful at a time on a terminal. It normally pauses after each screenful, printing "--More--" at the bottom of the screen. If the user then types a carriage return, one more line is displayed. If the user hits a space, another screenful is displayed. Other possibilities are described below.

The command line options are:

-n An integer which is the size (in lines) of the window which more will use instead of the default.

-c More will draw each page by beginning at the top of the screen and erasing each line just before it draws on it. This avoids scrolling the screen, making it easier to read while more is writing. This option will be ignored if the terminal does not have the ability to clear to the end of a line.

-d More will prompt the user with the message "Hit space to continue, Rubout to abort" at the end of each screenful. This is useful if more is being used as a filter in some setting, such as a class, where many users may be unsophisticated.

-f This causes more to count logical, rather than screen lines. That is, long lines are not folded. This option is recommended if nroff output is being piped through ul, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen positions, but that do not print when they are sent to the terminal as part of an escape sequence. Thus more may think that lines are longer than they actually are and fold lines erroneously.

-l Do not treat ^L (form feed) specially. If this option is not given, more will pause after any line that contains a ^L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

-s Squeeze multiple blank lines from the output, producing only one blank line. Especially helpful when viewing nroff output, this option maximizes the useful information present on the screen.

-u Normally, more will handle underlining such as produced by nroff in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a stand-out mode, more will output appropriate escape sequences to enable underlining or stand-out mode for underlined information in the source file. The -u option suppresses this processing.

-r Normally, more will eat control characters that it does not interpret in some way. The -r option will cause these to be displayed as ^C where C stands for any such character.

-w Normaly, more will exit when it comes to end of its input. -w will cause more to prompt and wait for any key to be struck before exiting.

+linenumber
Start up at linenumber.

+/pattern
Start up two lines before the line containing the regular expression pattern.

If the program is invoked as page, then the screen is cleared before each screenful is printed (but only if a full screenful is being printed), and k - 1 rather than k - 2 lines are printed in each screenful, where k is the number of lines the terminal can display.

More looks in the file /etc/termcap to determine terminal characteristics, and to determine the default window size. On a terminal capable of displaying 24 lines, the default window size is 22 lines.

More looks in the environment variable MORE to pre-set any flags desired. For example, if you prefer to view files using the -c mode of operation, the csh command setenv MORE -c or the sh command sequence MORE='-c' ; export MORE would cause all invocations of more , including invocations by programs such as man and msgs , to use this mode. Normally, the user will place the command sequence which sets up the MORE environment variable in the .cshrc or .profile file.

If more is reading from a file, rather than a pipe, then a percentage is displayed along with the "--More--" prompt. This gives the fraction of the file (in characters, not

lines) that has been read so far.

Other sequences which may be typed when more pauses, and their effects, are as follows (i is an optional integer argument, defaulting to 1) :

i<space>
display i more lines, (or another screenful if no argument is given)

^D display 11 more lines (a "scroll"). If i is given, then the scroll size is set to i.

d same as ^D (control-D)

iz same as typing a space except that i, if present, becomes the new window size.

is skip i lines and print a screenful of lines

if skip i screenfuls and print a screenful of lines

q or Q
Exit from more.

= Display the current line number.

v Start up the editor vi at the current line.

h or ?
Help command; give a description of all the more commands.

i/expr
search for the i-th occurrence of the regular expression expr. If there are less than i occurrences of expr, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

in search for the i-th occurrence of the last regular expression entered.

' (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!command
invoke a shell with command. The characters '%' and '!' in "command" are replaced with the current file name and the previous shell command respectively. If there is no current file name, '%' is not expanded. The sequences "\%" and "\!" are replaced by "%" and "!" respectively.

i:n skip to the i-th next file given in the command line (skips to last file if n doesn't make sense)

i:p skip to the i-th previous file given in the command line. If this command is given in the middle of printing out a file, then more goes back to the beginning of the file. If i doesn't make sense, more skips back to the first file. If more is not reading from a file, the bell is rung and nothing else happens.

:f display the current file name and line number.

:q or :Q
exit from more (same as q or Q).

. (dot) repeat the previous command.

The commands take effect immediately, i.e., it is not necessary to type a carriage return. Up to the time when the command character itself is given, the user may hit the line kill character to cancel the numerical argument being formed. In addition, the user may hit the erase character to redisplay the "--More--(xx%)" message.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\). More will stop sending output, and will display the usual "--More--" prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to noecho mode by this program so that the output can be continuous. What you type will thus not show on your terminal, except for the slash (/) and exclamation (!) commands.

If the standard output is not a teletype, then more acts just like cat, except that a header is printed before each file (if there is more than one).

A sample usage of more in previewing nroff output would be

```
nroff -ms +2 doc.n | more -s
```

FILES

/etc/termcap	Terminal data base
/usr/lib/more.help	Help file

SEE ALSO

csh(1S), man(1T), msgs(1), sh(1), environ(5)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NAME

mount - mount file system

SYNTAX

/etc/mount [special name [**-r**]]

DESCRIPTION

Mount announces to the system that a removable file system is present on the device special. The file name must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root. The optional last argument indicates that the file system is to be mounted read-only.

Umount performs the inverse dismounting operation, announcing to the system that the removable file system previously mounted on device special is to be removed. First, any pending I/O for the file system is completed, and the file system is flagged clean. Mount will refuse to mount a file system which is not flagged clean; this can happen if a system crash prevented the use of umount or haltsys(8). In such a case, use fsck(1M) to clean the file system, then try mount again.

These commands maintain a table of mounted devices. If invoked without an argument, mount prints the table.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

FILES

/etc/mtab: mount table

SEE ALSO

umount(1m) mount(2), mtab(5)

DIAGNOSTICS

Exit code 0 is returned for a successful mount, 1 for a failure, 2 for attempting to mount an unclean structure.

NOTES

Mounting file systems full of garbage will crash the system. Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

NAME

mv - move or rename files and directories

SYNTAX

mv filel file2

mv file ... directory

DESCRIPTION

Mv moves (changes the name of) filel to file2.

If file2 already exists, it is removed before filel is moved. If file2 has a mode which forbids writing, mv prints the mode (see chmod(2)) and reads the standard input to obtain a line; if the line begins with **y**, the move takes place; if not, mv exits.

In the second form, one or more files are moved to the directory with their original file-names.

Mv refuses to move a file onto itself.

SEE ALSO

cp(1), chmod(2), copy(1)

NOTES

If filel and file2 lie on different file systems, mv must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

Mv should take **-f** flag, like rm, to suppress the question if the target exists and is not writable.

NAME

ncheck - generate names from i-numbers

SYNTAX

ncheck [**-i** numbers] [-a] [-s] [filesystem]

DESCRIPTION

Ncheck with no argument generates a pathname vs. i-number list of all files on a set of default file systems. Names of directory files are followed by '/.'. The **-i** option reduces the report to only those files whose i-numbers follow. The -a option allows printing of the names '.' and '..', which are ordinarily suppressed. The -s option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

The report is in no useful order, and probably should be sorted.

SEE ALSO

dcheck(1), icheck(1), sort(1)

DIAGNOSTICS

When the filesystem structure is improper, '??' denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

NAME

neqn - format mathematics

SYNTAX

neqn [-dxy] [-fn] [file] ...
checkeq [file] ...

DESCRIPTION

Neqn is an nroff(1T) preprocessor for formatting mathematics on terminals and for printers; eqn(1T) is its counterpart for typesetting with troff(1T). Usage is almost always

neqn file ... | nroff

If no files are specified, these programs reads from the standard input. A line beginning with '.EQ' marks the start of an equation; the end of an equation is marked by a line beginning with '.EN'. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to set two characters as 'delimiters'; subsequent text between delimiters is also treated as neqn input. Delimiters may be set to characters x and y with the command-line argument **-d**xy or (more commonly) with 'delim xy' between .EQ and .EN. The left and right delimiters may be identical. Delimiters are turned off by 'delim off'. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched. Fonts can be changed globally in a document with **gfont** n, or with the command-line argument -fn.

The program checkeq reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within neqn are separated by spaces, tabs, newlines, braces, double quotes, tildes or circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character like x could appear, a complicated construction enclosed in braces may be used instead. Tilde ~ represents a full space in the output, circumflex ^ half as much.

SEE ALSO

eqn(1T), checkeq(1T), troff(1T), tbl(1T), ms(7), eqnchar(7)
B. W. Kernighan and L. L. Cherry, Typesetting Mathematics-User's Guide
J. F. Ossanna, NROFF/TROFF User's Manual

NOTES

To embolden digits, parens, etc., it is necessary to quote them, as in 'bold "12.3"'.

NAME

newgrp - log in to a new group

SYNTAX

newgrp group

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to login(1). The same person remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

A password is demanded if the group has a password and the user himself does not.

When most users log in, they are members of the group named 'other.' Newgrp is known to the shell, which executes it directly without a fork.

FILES

/etc/group, /etc/passwd

SEE ALSO

login(1), group(5)

NAME

nice - run a command at low priority

SYNTAX

nice [-number] command [arguments]

DESCRIPTION

Nice executes command with low scheduling priority. If the number argument is present, the priority is incremented (higher numbers mean lower priorities) by that amount up to a limit of 20. The default number is 10.

The super-user may run commands with priority higher than normal by using a negative priority, e.g. '--10'.

SEE ALSO

nohup(1), nice(2)

DIAGNOSTICS

Nice returns the exit status of the subject command.

NAME

nm - print name list

SYNTAX

nm [**-gnoprucvx**] [file ...]

DESCRIPTION

Nm prints the name list (symbol table) of each object file in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no file is given, the symbols in 'a.out' are listed.

Each symbol name is preceded by its value in hexadecimal (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), **C** (common symbol), or **K** (8086 common). If the symbol is local (non-external) the type letter is in lowercase. The output is sorted alphabetically.

Options are:

-g Print only global (external) symbols.

-n Sort numerically rather than alphabetically.

-o Prepend file or archive element name to each output line rather than only once.

-p Don't sort; print in symbol-table order.

-r Sort in reverse order.

-u Print only undefined symbols.

-c Print only C program symbols (symbols which begin with '_') as they appeared in the C program.

-v Also describe the object file and symbol table format.

If the object file is an 8086 relocatable file, this flag also causes three fields to be printed out after each symbol name that is not undefined. They are: a segment type descriptor, the type of relocation required and the (sequential) number of the segment to which it belongs. At the end of the namelist the segment table is printed out, including the segment number, the segment name, and the class name of each segment.

FILES

a.out Default input file.

SEE ALSO
ar(1S), ar(5), a.out(5)

NAME

nohup - run a command immune to hangups and quits

SYNTAX

nohup command [arguments]

DESCRIPTION

Nohup executes command with hangups and quits ignored. Use it to place a command in the background while logged off. If output is not re-directed by the user, it will be sent to **nohup.out**. If **nohup.out** is not writable in the current directory, output is redirected to **$HOME/nohup.out**.

SEE ALSO

nice(1), signal(2).

NAME

nroff - text formatter

SYNTAX

nroff [option] ... [file] ...

DESCRIPTION

Nroff formats text in the named files. Troff is part of the nroff/troff family of text formatters. Nroff is used to format files for output to a lineprinter or daisy wheel printer; troff to a Graphic Systems C/A/T phototypesetter.

If no file argument is present, the standard input is read. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input. The options, which may appear in any order so long as they appear before the files, are:

-olist Print only pages whose page numbers appear in the comma-separated list of numbers and ranges. A range N-M means pages N through M; an initial -N means from the beginning to page N; and a final N- means from N to the end.

-nN Number first generated page N.

-sN Stop every N pages. Nroff will halt prior to every N pages (default N=1) to allow paper loading or changing, and will resume upon receipt of a newline.

-mname Prepend the macro file /usr/lib/tmac/tmac.name to the input files.

-raN Set register a (one-character) to N.

-i Read standard input after the input files are exhausted.

-q Invoke the simultaneous input-output mode of the .rd request.

-Tname Prepare output for specified terminal. Known names are **37** for the (default) Teletype Corporation Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

-e Produce equally-spaced words in adjusted lines, using full terminal resolution.

-h Use output tabs during horizontal spacing to speed

output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables

SEE ALSO

J. F. Ossanna, Nroff/Troff user's manual
B. W. Kernighan, A TROFF Tutorial (Reg.).br col(1T), eqn(1T), tbl(1T), troff(1T)

NAME

od - octal dump

SYNTAX

od [-bcdox] [file] [[+]offset[.][b]]

DESCRIPTION

Od dumps file in one or more formats as selected by the first argument. If the first argument is missing, **-o** is default. The meanings of the format argument characters are:

b Interpret bytes in octal.

c Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=\0, backspace=\b, formfeed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers.

d Interpret words in decimal.

o Interpret words in octal.

x Interpret words in hex.

The file argument specifies which file is to be dumped. If no file argument is specified, the standard input is used.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If '.' is appended, the offset is interpreted in decimal. If 'b' is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded '+'.

Dumping continues until end-of-file.

SEE ALSO

adb(1S)

NAME

passwd - change login password

SYNTAX

passwd [name]

DESCRIPTION

This command changes (or installs) a password associated with the user name (your own name by default).

The program prompts for the old password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes.

New passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. These rules are relaxed if you are insistent enough.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(5), crypt(3)
Robert Morris and Ken Thompson, Password Security: A Case History

NAME

plot - graphics filters

SYNTAX

plot [-Tterminal [raster]]

DESCRIPTION

These commands read plotting instructions (see plot(5)) from the standard input, and in general produce plotting instructions suitable for a particular terminal on the standard output.

If no terminal type is specified, the environment parameter $TERM (see environ(5)) is used. Known terminals are:

4014 Tektronix 4014 storage scope.

450 DASI Hyterm 450 terminal (Diablo mechanism).

300 DASI 300 or GSI terminal (Diablo mechanism).

300S DASI 300S terminal (Diablo mechanism).

ver Versatec D1200A printer-plotter. This version of plot places a scan-converted image in '/usr/tmp/raster' and sends the result directly to the plotter device rather than to the standard output. The optional argument causes a previously scan-converted file raster to be sent to the plotter.

FILES

/usr/bin/tek
/usr/bin/t450
/usr/bin/t300
/usr/bin/t300s
/usr/bin/vplot
/usr/tmp/raster

SEE ALSO

plot(3), plot(5)

NOTES

There is no lockout protection for /usr/tmp/raster.

NAME

pr - print file

SYNTAX

pr [option] ... [file] ...

DESCRIPTION

Pr produces a printed listing of one or more files. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, pr prints its standard input.

Options apply to all following files but may be reset between files:

-n Produce n-column output.

+n Begin printing with page n.

-h Take the next argument as a page header.

-wn For purposes of multi-column output, take the width of the page to be n characters instead of the default 72.

-ln Take the length of the page to be n lines instead of the default 66.

-t Do not print the 5-line header or the 5-line trailer normally supplied for each page.

-sc Separate columns by the single character c instead of by the appropriate amount of white space. A missing c is taken to be a tab.

-m Print all files simultaneously, each in one column,

-b A form feed character is used to separate pages, normally pages are separated by a number of newline characters.

Inter-terminal messages via write(1) are forbidden during a pr.

FILES

/dev/tty* to suspend messages.

SEE ALSO

cat(1)

DIAGNOSTICS

There are no diagnostics when pr is printing on a terminal.

NAME

prep - prepare text for statistical processing

SYNTAX

prep [-dio] file ...

DESCRIPTION

Prep reads each file in sequence and writes it on the standard output, one 'word' to a line. A word is a string of alphabetic characters and imbedded apostrophes, delimited by space or punctuation. Hyphented words are broken apart; hyphens at the end of lines are removed and the hyphenated parts are joined. Strings of digits are discarded.

The following option letters may appear in any order:

-d Print the word number (in the input stream) with each word.

-i Take the next file as an 'ignore' file. These words will not appear in the output. (They will be counted, for purposes of the -d count.)

-o Take the next file as an 'only' file. Only these words will appear in the output. (All other words will also be counted for the -d count.)

-p Include punctuation marks (single nonalphanumeric characters) as separate output lines. The punctuation marks are not counted for the -d count.

Ignore and only files contain words, one per line.

SEE ALSO

deroff(1T)

NAME

printenv - print out the environment

SYNTAX

printenv [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a name is specified, only its value is printed.

If a name is specified and it is not defined in the environment, printenv returns exit status 1, else it returns status 0.

SEE ALSO

sh(1), environ(5), csh(1S)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NAME

prof - display profile data

SYNTAX

prof [**-v**] [**-a**] [**-l**] [-low [-high]] [file]

DESCRIPTION

Prof interprets the file mon.out produced by the monitor subroutine. Under default modes, the symbol table in the named object file (a.out default) is read and correlated with the mon.out profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **-a** option is used, all symbols are reported rather than just external symbols. If the **-l** option is used, the output is listed by symbol value rather than decreasing percentage.

If the **-v** option is used, all printing is suppressed and a graphic version of the profile is produced on the standard output for display by the plot(1) filters. The numbers low and high, by default 0 and 100, cause a selected percentage of the profile to be plotted with accordingly higher resolution.

FILES

mon.out for profile
a.out for namelist

SEE ALSO

monitor(3), profil(2), cc(1S), plot(1)

NOTES

Beware of quantization errors.

If you use an explicit call to monitor(3) you will need to make sure that the buffer size is equal to or smaller than the program size.

NAME

ps - process status

SYNTAX

ps [**alx**] [namelist]

DESCRIPTION

Ps prints certain indicia about active processes. The **a** option asks for information about all processes with terminals (ordinarily only one's own processes are displayed); **x** asks even about processes with no terminal; **l** asks for a long listing. The short listing contains the process ID, tty name, the cumulative execution time of the process and an approximation to the command line.

The long listing is columnar and contains

F Flags associated with the process. 01: in core; 02: system process; 04: locked in core (e.g. for physical I/O); 10: being swapped; 20: being traced by another process.

S The state of the process. 0: nonexistent; S: sleeping; W: waiting; R: running; I: intermediate; Z: terminated; T: stopped.

UID The user ID of the process owner.

PID The process ID of the process; if you know the true name of a process it is possible to kill the process with the kill command.

PPID The process ID of the parent process.

CPU Processor utilization for scheduling.

PRI The priority of the process; high numbers mean low priority.

NICE Used in priority computation.

ADDR The core address of the process if resident, otherwise the disk address.

SZ The size in blocks of the core image of the process.

ICT If the process is currently resides in core memory, then ICT (in-core time) represents the number of seconds that the process has been resident, up to a limit of 127. If the process is currently on the swapper, then ICT is the number of seconds since the process was swapped out of core.

WCHAN
The event for which the process is waiting or sleeping; if blank, the process is running.

TTY The controlling tty for the process.

TIME The cumulative execution time for the process.

COMMAND
The command and its arguments.

A process that has exited and has a parent, but has not yet been waited for by the parent is marked <defunct>. Ps makes an educated guess as to the file name and arguments given when the process was created by examining core memory or the swap area. The method is inherently somewhat unreliable and in any event a process is entitled to destroy this information, so the names cannot be counted on too much.

If a second argument is given, it is taken to be the file containing the system's namelist.

FILES

/xenix	system namelist
/dev/mem	core memory
/dev	searched to find swap device and tty names

SEE ALSO
kill(1)

NOTES
Things can change while ps is running; the picture it gives is only a close approximation to reality.
Some data printed for defunct processes is irrelevant

NAME

pstat - print system facts

SYNTAX

pstat [**-aixpuf**] [suboptions] [file]

DESCRIPTION

Pstat interprets the contents of certain system tables. If file is given, the tables are sought there, otherwise in /dev/mem. The required namelist is taken from /xenix. Options are

-a Under -p, describe all process slots rather than just active ones.

-i Print the inode table with the these headings:

LOC The core location of this table entry.
FLAGS Miscellaneous state variables encoded thus:
- L locked
- U update time filsys(5)) must be corrected
- A access time must be corrected
- M file system is mounted here
- W wanted by another process (L flag is on)
- T contains a text file
- C changed time must be corrected

CNT Number of open file table entries for this inode.
DEV Major and minor device number of file system in which this inode resides.
INO I-number within the device.
MODE Mode bits, see chmod(2).
NLK Number of links to this inode.
UID User ID of owner.
SIZ/DEV
Number of bytes in an ordinary file, or major and minor device of special file.

-x Print the text table with these headings:

LOC The core location of this table entry.
FLAGS Miscellaneous state variables encoded thus:
- T ptrace(2) in effect
- W text not yet written on swap device
- L loading in progress
- K locked
- w wanted (L flag is on)

DADDR Disk address in swap, measured in multiples of BLOCK-SIZE bytes.

CADDR Core address, measured in units of memory management resolution, usually either 64 or 128 bytes.

SIZE Size of text segment, measured in units of memory management resolution, usually either 64 or 128 bytes.

IPTR Core location of corresponding inode.

CNT Number of processes using this text segment.

CCNT Number of processes in core using this text segment.

-p Print process table for active processes with these headings:

LOC The core location of this table entry.
S Run state encoded thus:
0 no process
1 waiting for some event
3 runnable
4 being created
5 being terminated
6 stopped under trace
F Miscellaneous state variables, OR-ed together:
01 loaded
02 the scheduler process
04 locked
010 swapped out
020 traced
040 used in tracing
0100 locked in by lock(2).
PRI Scheduling priority, see nice(2).
SIGNAL
Signals received (signals 1-16 coded in bits 0-15),
UID Real user ID.
TIM Time resident in seconds; times over 127 coded as 127.
CPU Weighted integral of CPU time, for scheduler.
NI Nice level, see nice(2).
PGRP Process number of root of process group (the opener of the controlling terminal).
PID The process ID number.
PPID The process ID of parent process.
ADDR If in core, the physical address of the 'u-area' of the process measured in units of memory management resolution, usually either 64 or 128 bytes. If swapped out, the position in the swap area measured in multiples of BLOCKSIZE bytes.
SIZE Size of process image, measured in units of memory management resolution, usually usually 64 or 128 bytes.
WCHAN Wait channel number of a waiting process.
LINK Link pointer in list of runnable processes.
TEXTP If text is pure, pointer to location of text table entry.
CLKT Countdown for alarm(2) measured in seconds.

-u print information about a user process; the next argument is its address as given by ps(1). The process must be in main memory, or the file used can be a core image and the address 0.

-f Print the open file table with these headings:

LOC The core location of this table entry.
FLG Miscellaneous state variables encoded thus:
R open for reading
W open for writing
P pipe
CNT Number of processes that know this open file.
INO The location of the inode table entry for this file.
OFFS The file offset, see lseek(2).

FILES

/xenix namelist
/dev/mem default source of tables

SEE ALSO

ps(1), stat(2), filsys(5)
K. Thompson, UNIX Implementation

NAME

ptx - permuted index

SYNTAX

ptx [option] ... [input [output]]

DESCRIPTION

Ptx generates a permuted index to file input on file output (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of the page. Ptx produces output in the form:

.xx "tail" "before keyword" "keyword and after" "head"

where .xx may be an nroff or troff(1T) macro for user-defined formatting. The before keyword and keyword and after fields incorporate as much of the line as will fit around the keyword when it is printed at the middle of the page. Tail and head, at least one of which is an empty string "", are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line. When original text must be discarded, '/' marks the spot.

The following options can be applied:

-f Fold upper and lower case letters for sorting.

-t Prepare the output for the phototypesetter; the default line length is 100 characters.

-w n Use the next argument, n, as the width of the output line. The default line length is 72 characters.

-g n Use the next argument, n, as the number of characters to allow for each gap among the four parts of the line as finally printed. The default gap is 3 characters.

-o only
Use as keywords only the words given in the only file.

-i ignore
Do not use as keywords any words given in the ignore file. If the -i and -o options are missing, use /usr/lib/eign as the ignore file.

-b break
Use the characters in the break file to separate words. In any case, tab, newline, and space characters are always used as break characters.

-r Take any leading nonblank characters of each input line to be a reference identifier (as to a page or chapter) separate from the text of the line. Attach that identifier as a 5th field on each output line.

The index for this manual was generated using ptx.

FILES

/bin/sort
/usr/lib/eign

NOTES

Line length counts do not account for overstriking or proportional spacing.

NAME

pwd - working directory name

SYNTAX

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory. Use pwd when you've lost track of where you are in the directory hierarchy.

SEE ALSO

cd(1)

NAME

quot - summarize file system ownership

SYNTAX

quot [option] ... [filesystem]

DESCRIPTION

Quot prints the number of blocks in the named filesystem currently owned by each user. If no filesystem is named, a default name is assumed. The following options are available:

-n Cause the pipeline

```
ncheck filesystem | sort +0n | quot -n filesystem
```

to produce a list of all files and their owners.

-c Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.

-f Print count of number of files as well as space owned by each user.

FILES

Default file system varies with system.
/etc/passwd to get user names

SEE ALSO

ls(1), du(1)

NOTES

Holes in files are counted as if they actually occupied space.

NAME

random - generate a random number

SYNTAX

random [options] [seed]

DESCRIPTION

This command generates a random number as it's exit value after reading from the standard input. The number is in the range 0 to (2^15)-1. The single integer argument seed can be given as a key from which the random number is computed. The **-e** option causes random to exit immediately without reading from the standard input.

SEE ALSO

rand(3), srand(3)

NAME

ranlib - convert archives to random libraries

SYNTAX

ranlib archive ...

DESCRIPTION

Ranlib converts each archive to a form which can be loaded more rapidly by the loader, by adding a table of contents named **__.SYMDEF** to the beginning of the archive. It uses ar(1S) to reconstruct the archive, so that sufficient temporary file space must be available in the file system containing the current directory.

SEE ALSO

ld(1S), ar(1S), copy(1), settime(1)

NOTES

Because generation of a library by ar and randomization by ranlib are separate, phase errors are possible. The loader ld warns when the modification date of a library is more recent than the creation of its dictionary; but this means you get the warning even if you only copy the library.

NAME

ratfor - rational Fortran dialect

SYNTAX

ratfor [option ...] [filename ...]

DESCRIPTION

Ratfor converts a rational dialect of Fortran into ordinary irrational Fortran. Ratfor provides control flow constructs essentially identical to those in C:

```
statement grouping:
     { statement; statement; statement }

decision-making:
     if (condition) statement [ else statement ]
     switch (integer value) {
          case integer:  statement
          ...
          [ default: ]   statement
     }

loops:
     while (condition) statement
     for (expression; condition; expression) statement
     do limits statement
     repeat statement [ until (condition) ]
     break [n]
     next [n]
```

and some syntactic sugar to make programs easier to read and write:

```
free form input:
     multiple statements/line; automatic continuation

comments:
     # this is a comment

translation of relationals:
     >, >=, etc., become .GT., .GE., etc.

return (expression)
     returns expression to caller from function

define:
     define name replacement

include:
     include filename
```

The option **-h** causes quoted strings to be turned into 27H constructs. **-C** copies comments to the output, and attempts

to format it neatly. Normally, continuation lines are marked with a & in column 1; the option **-6x** makes the continuation character **x** and places it in column 6.

SEE ALSO

struct(1S)
B. W. Kernighan and P. J. Plauger, Software Tools, Addison-Wesley, 1976.

NAME

refer, lookbib - find and insert literature references in documents

SYNTAX

refer [option] ...

lookbib [file] ...

DESCRIPTION

Lookbib accepts keywords from the standard input and searches a bibliographic data base for references that contain those keywords anywhere in title, author, journal name, etc. Matching references are printed on the standard output. Blank lines are taken as delimiters between queries.

Refer is a preprocessor for nroff or troff(1T) that finds and formats references. The input files (standard input default) are copied to the standard output, except for lines between .[and .] command lines, which are assumed to contain keywords as for lookbib, and are replaced by information from the bibliographic data base. The user may avoid the search, override fields from it, or add new fields. The reference data, from whatever source, are assigned to a set of troff strings. Macro packages such as ms(7) print the finished reference text from these strings. A flag is placed in the text at the point of reference; by default the references are indicated by numbers.

The following options are available:

-a*r* Reverse the first *r* author names (Jones, J. A. instead of J. A. Jones). If *r* is omitted all author names are reversed.

-b Bare mode: do not put any flags in text (neither numbers nor labels).

-c*string*
Capitalize (with CAPS SMALL CAPS) the fields whose key-letters are in *string*.

-e Instead of leaving the references where encountered, accumulate them until a sequence of the form

```
.[
$LIST$
.]
```

is encountered, and then write out all references collected so far. Collapse references to the same source.

-k*x* Instead of numbering references, use labels as

specified in a reference data line beginning %x; by default x is **L**.

-lm,n Instead of numbering references, use labels made from the senior author's last name and the year of publication. Only the first m letters of the last name and the last n digits of the date are used. If either m or ,n is omitted the entire name or date respectively is used.

-p Take the next argument as a file of references to be searched. The default file is searched last.

-n Do not search the default file.

-skeys
Sort references by fields whose key-letters are in the keys string; permute reference numbers in text accordingly. Implies **-e**. The key-letters in keys may be followed by a number to indicate how many such fields are used, with + taken as a very large number. The default is **AD** which sorts on the senior author and then date; to sort, for example, on all authors and then title use **-sA+T**.

To use your own references, put them in the format described in pubindex(1T) They can be searched more rapidly by running pubindex(1T) on them before using refer; failure to index results in a linear search.

When refer is used with eqn, neqn or tbl, refer should be first, to minimize the volume of data passed through pipes.

FILES

/usr/dict/papers directory of default publication lists and indexes
/usr/lib/refer directory of programs

SEE ALSO

nroff(1T), troff(1T)

NAME

restor - incremental file system restore

SYNTAX

restor key [argument ...]

DESCRIPTION

Restor is used to read magtapes dumped with the dump command. The key specifies what is to be done. Key is one of the characters rRxt optionally combined with f.

f Use the first argument as the name of the tape instead of the default.

r or R
The tape is read and loaded into the file system specified in argument. This should not be done lightly (see below). If the key is R restor asks which tape of a multi volume set to start on. This allows restor to be interrupted and then restarted (an icheck -s must be done before restart).

x Each file on the tape named by an argument is extracted. The file name has all 'mount' prefixes removed; for example, /usr/bin/lpr is named /bin/lpr on the tape. The file extracted is placed in a file with a numeric name supplied by restor (actually the inode number). In order to keep the amount of tape read to a minimum, the following procedure is recommended:

Mount volume 1 of the set of dump tapes.

Type the restor command.

Restor will announce whether or not it found the files, give the number it will name the file, and rewind the tape.

It then asks you to 'mount the desired tape volume'. Type the number of the volume you choose. On a multi volume dump the recommended procedure is to mount the last through the first volume in that order. Restor checks to see if any of the files requested are on the mounted tape (or a later tape, thus the reverse order) and doesn't read through the tape if no files are. If you are working with a single volume dump or the number of files being restored is large, respond to the query with '1' and restor will read the tapes in sequential order.

If you have a hierarchy to restore you can use dumpdir(1) to produce the list of names and a shell script

to move the resulting files to their homes.

t Print the date the tape was written and the date the filesystem was dumped from.

The r option should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape onto this. Thus

```
/etc/mkfs /dev/rp0 40600
restor r /dev/rp0
```

is a typical sequence to restore a complete dump, where rp0 is the name of a disk device. Another restor can be done to get an incremental dump in on top of this.

A dump followed by a mkfs and a restor is used to change the size of a file system.

FILES

default tape unit varies with installation
rst*

SEE ALSO

dump(1), mkfs(1), dumpdir(1)

DIAGNOSTICS

There are various diagnostics involved with reading the tape and writing the disk. There are also diagnostics if the i-list or the free list of the file system is not large enough to hold the dump.

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

NOTES

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, restor doesn't use it.

NAME

rev - reverse lines of a file

SYNTAX

rev [file] ...

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

NAME

rm - remove (unlink) files

SYNTAX

rm [-fri] file ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to a file, then the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with 'y' the file is deleted, otherwise the file remains. No questions are asked when the **-f** (force) option is given.

If a designated file is a directory, an error comment is printed unless the optional argument -r has been used. In that case, rm recursively deletes the entire contents of the specified directory, and the directory itself.

If the **-i** (interactive) option is in effect, rm asks whether to delete each file, and, under -r, whether to examine each directory.

SEE ALSO

rmdir(1), unlink(2)

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file '..' to avoid the consequences of inadvertently doing something like "rm -r .*".

NAME

rmdir - remove (unlink) files

SYNTAX

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more subdirectories from a directory. A directory must be empty before it can be removed. Rmdir enforces a standard and safe procedure for removing a directory; that is, first delete the contents of the directory and only then remove the directory itself. Note that the "rm -r dir" command is a more dangerous alternative to rmdir.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

rm(1) unlink(2)

DIAGNOSTICS

Generally self-explanatory.

NAME

rmuser - remove a user from the system

SYNTAX

/etc/rmuser

DESCRIPTION

Rmuser removes users from the system. It begins by prompting for a user name; after receiving a valid user name as a response, it then deletes the named user's entry in the password file, and removes the user's mail box file, the .profile file, and the entire home directory.

Before removing a user ID from the system, make sure its mail box is empty and that all files belonging to that user ID have been saved or deleted as required.

The rmuser program will refuse to remove a user ID or any of its files if one or more of the following checks fails:

- The user name given is one of the "system" user names such as root, sys, sysinfo, cron, or uucp. All user ID's less than 200 are considered reserved for system use, and cannot be removed using rmuser.

- The user's mail box exists and is not èmpty.

- The user's home directory contains files other than .profile.

Rmuser can only be executed by the super-user.

FILES

/etc/passwd
/usr/spool/mail/username.
$HOME

SEE ALSO

mkuser(1), dump(1)

NAME

sa - system accounting

SYNTAX

sa [-abcijlnrstuv] [file]

DESCRIPTION

Sa reports on, cleans up, and generally maintains accounting files. System accounting must first be turned on with accton . The command "accton file , where file is an existing file, causes system accounting information for every process executed to be placed at the end file. If no argument is given, accounting is turned off.

Sa is able to condense the information in /usr/adm/acct into a summary file /usr/adm/savacct which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system acct can grow by 100 blocks per day. The summary file is read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; /usr/adm/acct is the default. There are zillions of options:

a Place all command names containing unprintable characters and those used only once under the name '***other.'

b Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.

c Besides total user, system, and real time for each command print percentage of total time over all commands.

i Ignore the summary files /usr/adm/savacct and /usr/adm/usracct; do not include their contents in this report.

j Instead of total minutes time for each category, give seconds per call.

l Separate system and user time; normally they are combined.

m Print number of processes and number of CPU minutes for each user.

n Sort by number of calls.

r Reverse order of sort.

s Merge accounting file into summary file /usr/adm/savacct when done.

t For each command, report ratio of real time to the sum of user and system times.

u Superseding all other flags, print for each command in the accounting file the user ID and command name.

v If the next character is a digit n, then type the name of each command used n times or fewer. Await a reply from the typewriter; if it begins with 'y', add the command to the category '**junk**.' This is used to strip out garbage.

(default)
A table of 4 columns is printed: the number of calls, the total real time, the total combined system and user time, and the name of the command. The first line in the table contains the sum of each column.

FILES

/usr/adm/acct raw accounting
/usr/adm/savacct summary
/usr/adm/usracct per-user summary

SEE ALSO

ac(1), acct(2), accton(1m)

NAME

sddate - print and set dump dates

SYNTAX

sddate [name lev date]

DESCRIPTION

If no argument is given, the contents of the dump date file '/etc/ddate' are printed. The dump date file is maintained by dump(1M) and contains the date of the most recent dump for each dump level for each filesystem.

If arguments are given, an entry is replaced or made in '/etc/ddate'. name is the last component of the device pathname. lev is the dump level number (from 0 to 9), and date is a time in the form taken by date(1).

Some sites may wish to backup filesystems by coping them verbatim to dismountable packs. Sddate could be used to make a 'level 0' entry in '/etc/ddate', which would then allow incremental mag tape dumps.

For example:

```
sddate rrp3 5 10081520
```

makes an '/etc/ddate' entry showing a level 5 dump of '/dev/rrp3' on October 8, at 3:20 PM.

FILES

/etc/ddate

SEE ALSO

dump(1M), date(1)

DIAGNOSTICS

'bad conversion' if the date set is syntactically incorrect.

NAME

sed - stream editor

SYNTAX

sed [**-n**] [**-e** script] [-f sfile] [file] ...

DESCRIPTION

Sed copies the named files (standard input default) to the standard output, edited according to a script of commands. The -f option causes the script to be taken from file sfile; these options accumulate. If there is just one **-e** option and no -f's, the flag **-e** may be omitted. The **-n** option suppresses the default output.

A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation sed cyclically copies a line of input into a pattern space (unless there is something left after a 'D' command), applies in sequence all commands whose addresses select that pattern space, and at the end of the script copies the pattern space to the standard output (except under **-n**) and deletes the pattern space.

An address is either a decimal number that counts input lines cumulatively across files, a '$' that addresses the last line of input, or a context address, '/regular expression/', in the style of ed(1) modified thus:

The escape sequence '\n' matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function '!' (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

An argument denoted text consists of one or more lines, all but the last of which end with '\' to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an 's' command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument denoted rfile or wfile must terminate the command line and must be preceded by exactly one blank. Each wfile is created before processing begins. There can be at most 10 distinct wfile arguments.

(1)a\
text
Append. Place text on the output before reading the next input line.

(2)b label
Branch to the ':' command bearing the label. If label is empty, branch to the end of the script.

(2)c\
text
Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place text on the output. Start the next cycle.

(2)d Delete the pattern space. Start the next cycle.

(2)D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2)g Replace the contents of the pattern space by the contents of the hold space.

(2)G Append the contents of the hold space to the pattern space.

(2)h Replace the contents of the hold space by the contents of the pattern space.

(2)H Append the contents of the pattern space to the hold space.

(1)i\
text Insert. Place text on the standard output.

(2)l List the pattern space on the standard output in an

unambiguous form. Non-printing characters are spelled in two digit ascii, and long lines are folded.

(2)n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2)N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2)p Print. Copy the pattern space to the standard output.

(2)P Copy the initial segment of the pattern space through the first newline to the standard output.

(1)q Quit. Branch to the end of the script. Do not start a new cycle.

(2)r rfile
Read the contents of rfile. Place them on the output before reading the next input line.

(2)s/regular expression/replacement/flags
Substitute the replacement string for instances of the regular expression in the pattern space. Any character may be used instead of '/'. For a fuller description see ed(1). Flags is zero or more of

g Global. Substitute for all nonoverlapping instances of the regular expression rather than just the first one.

p Print the pattern space if a replacement was made.

w wfile
Write. Append the pattern space to wfile if a replacement was made.

(2)t label
Test. Branch to the ':' command bearing the label if any substitutions have been made since the most recent reading of an input line or execution of a 't'. If label is empty, branch to the end of the script.

(2)w wfile
Write. Append the pattern space to wfile.

(2)x Exchange the contents of the pattern and hold spaces.

(2)y/string1/string2/
Transform. Replace all occurrences of characters in string1 with the corresponding character in string2.

The lengths of string1 and string2 must be equal.

(2)! function
Don't. Apply the function (or group, if function is '{') only to lines not selected by the address(es).

(0): label
This command does nothing; it bears a label for 'b' and 't' commands to branch to.

(1)= Place the current line number on the standard output as a line.

(2){ Execute the following commands through a matching '}' only when the pattern space is selected.

(0) An empty command is ignored.

SEE ALSO

ed(1), grep(1), awk(1)

NAME

settime - change the access and modification dates of files

SYNTAX

settime [yymmddhhmm [.ss]] [**-f** fname] name ...

DESCRIPTION

Set the access and modification dates for one or more files. The dates are set to the specified date, or to the access and modification dates of the file specified via **-f**. Exactly one of these methods must be used to specify the new date(s). yy is the last two digits of the year; the first mm is the month number; dd is the day number in the month; hh is the hour number (24 hour system); the second mm is the minute number; .ss is optional and is the seconds. For example:

settime 10080045 ralph pete

sets the access and modification dates of files ralph and pete to Oct 8, 12:45 AM. The year, month and day may be omitted, the current values being the current date.

settime -f ralph john

sets the access and modification dates of the file john to those of the file ralph.

NAME

sh, for, case, if, while, :, ., break, continue, cd, eval, exec, exit, export, login, newgrp, read, readonly, set, shift, times, trap, umask, wait - command language

SYNTAX

sh [-ceiknrstuvx] [arg] ...

DESCRIPTION

Sh is a command programming language that executes commands read from a terminal or a file. See invocation for the meaning of arguments to the shell.

Commands.

A simple-command is a sequence of non blank words separated by blanks (a blank is a tab or a space). The first word specifies the name of the command to be executed. Except as specified below the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see exec(2)). The value of a simple-command is its exit status if it terminates normally or 200+status if it terminates abnormally (see signal(2) for a list of status values).

A pipeline is a sequence of one or more commands separated by |. The standard output of each command but the last is connected by a pipe(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

A list is a sequence of one or more pipelines separated by ;, &, && or || and optionally terminated by ; or &. ; and & have equal precedence which is lower than that of && and ||, && and || also have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding pipeline to be executed without waiting for it to finish. The symbol && (||) causes the list following to be executed only if the preceding pipeline returns a zero (non zero) value. Newlines may appear in a list, instead of semicolons, to delimit commands.

A command is either a simple-command or one of the following. The value returned by a command is that of the last simple-command executed in the command.

for name [in word ...] do list done

Each time a for command is executed name is set to the next word in the for word list If in word ... is omitted then in "$@" is assumed. Execution ends when there are no more words in the list.

case word in [pattern [| pattern] ...) list ;;] ... esac

A **case** command executes the list associated with the first pattern that matches word. The form of the patterns is the same as that used for file name generation.

if list **then** list [**elif** list **then** list] ... [**else** list] **fi**
The list following **if** is executed and if it returns zero the list following **then** is executed. Otherwise, the list following **elif** is executed and if its value is zero the list following **then** is executed. Failing that the **else** list is executed.

while list [**do** list] **done**
A **while** command repeatedly executes the **while** list and if its value is zero executes the **do** list; otherwise the loop terminates. The value returned by a **while** command is that of the last executed command in the **do** list. **until** may be used in place of **while** to negate the loop termination test.

(list)
Execute list in a subshell.

{ list }
list is simply executed.

The following words are only recognized as the first word of a command and when not quoted.

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

Command substitution.
The standard output from a command enclosed in a pair of grave accents (``) may be used as part or all of a word; trailing newlines are removed.

Parameter substitution.
The character **$** is used to introduce substitutable parameters. Positional parameters may be assigned values by **set**. Variables may be set by writing

name=value [name=value] ...

${parameter}
A parameter is a sequence of letters, digits or underscores (a name), a digit, or any of the characters *** @ # ? - $!**. The value, if any, of the parameter is substituted. The braces are required only when parameter is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If parameter is a digit then it is a positional parameter. If

parameter is * or @ then all the positional parameters, starting with $1, are substituted separated by spaces. $0 is set from argument zero when the shell is invoked.

${parameter-word}
If parameter is set then substitute its value; otherwise substitute word.

${parameter=word}
If parameter is not set then set it to word; the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${parameter?word}
If parameter is set then substitute its value; otherwise, print word and exit from the shell. If word is omitted then a standard message is printed.

${parameter+word}
If parameter is set then substitute word; otherwise substitute nothing.

In the above word is not evaluated unless it is to be used as the substituted string. (So that, for example, echo ${d-`pwd`} will only execute pwd if d is unset.)

The following parameters are automatically set by the shell.

- **#** The number of positional parameters in decimal.
- **-** Options supplied to the shell on invocation or by **set**.
- **?** The value returned by the last executed command in decimal.
- **$** The process number of this shell.
- **!** The process number of the last background command invoked.

The following parameters are used but not set by the shell.

- **HOME** The default argument (home directory) for the cd command.
- **PATH** The search path for commands (see **execution**).
- **MAIL** If this variable is set to the name of a mail file then the shell informs the user of the arrival of mail in the specified file.
- **PS1** Primary prompt string, by default '$ '.
- **PS2** Secondary prompt string, by default '> '.
- **IFS** Internal field separators, normally **space**, **tab**, and **newline**.

Blank interpretation.
After parameter and command substitution, any results of

substitution are scanned for internal field separator characters (those found in **$IFS**) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.

File name generation.
Following substitution, each command word is scanned for the characters *****, **?** and **[**. If one of these characters appears then the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern then the word is left unchanged. The character **.** at the start of a file name or immediately following a **/**, and the character **/**, must be matched explicitly.

***** Matches any string, including the null string.
? Matches any single character.
[...]
Matches any one of the characters enclosed. A pair of characters separated by **-** matches any character lexically between the pair.

Quoting.
The following characters have a special meaning to the shell and cause termination of a word unless quoted.

; & ( ) | < > newline space tab

A character may be quoted by preceding it with a ****. **\newline** is ignored. All characters enclosed between a pair of quote marks (''), except a single quote, are quoted. Inside double quotes ("") parameter and command substitution occurs and **** quotes the characters **\ ` " $**.

"$*" is equivalent to **"$1 $2 ..."** whereas
"$@" is equivalent to **"$1" "$2" ...** .

Prompting.
When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a newline is typed and further input is needed to complete a command then the secondary prompt (**$PS2**) is issued.

Input output.
Before a command is executed its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a command and are not passed on to the invoked command. Substitution occurs before word or digit is used.

<word
Use file word as standard input (file descriptor 0).

>word
Use file word as standard output (file descriptor 1). If the file does not exist then it is created; otherwise it is truncated to zero length.

>>word
Use file word as standard output. If the file exists then output is appended (by seeking to the end); otherwise the file is created.

<<word
The shell input is read up to a line the same as word, or end of file. The resulting document becomes the standard input. If any character of word is quoted then no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, \newline is ignored, and \ is used to quote the characters \ $ ` and the first character of word.

<&digit
The standard input is duplicated from file descriptor digit; see dup(2). Similarly for the standard output using >.

<&- The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit then the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example,

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

creates file descriptor 2 to be a duplicate of file descriptor 1.

If a command is followed by & then the default standard input for the command is the empty file (/dev/null). Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input output specifications.

Environment.
The environment is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list; see exec(2) and environ(5). The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name

found, giving it the corresponding value. Executed commands inherit the same environment. If the user modifies the values of these parameters or creates new ones, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any simple-command may be augmented by prefixing it with one or more assignments to parameters. Thus these two lines are equivalent

```
TERM=450 cmd args
(export TERM; TERM=450; cmd args)
```

If the **-k** flag is set, all keyword arguments are placed in the environment, even if the occur after the command name. The following prints 'a=b c' and 'c':

```
echo a=b c
set -k
echo a=b c
```

Signals.
The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the shell from its parent. (But see also **trap.**)

Execution.
Each time a command is executed the above substitutions are carried out. Except for the 'special commands' listed below a new process is created and an attempt is made to execute the command via an exec(2).

The shell parameter **$PATH** defines the search path for the directory containing the command. Each alternative directory name is separated by a colon (:). The default path is **:/bin:/usr/bin.** If the command name contains a / then the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an a.out file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a subshell.

Special commands.
The following commands are executed in the shell process and except where specified no input output redirection is permitted for such commands.

: No effect; the command does nothing.

. file
Read and execute commands from file and return. The search path **$PATH** is used to find the directory containing file.

break [n]
Exit from the enclosing **for** or **while** loop, if any. If n is specified then break n levels.

continue [n]
Resume the next iteration of the enclosing **for** or **while** loop. If n is specified then resume at the n-th enclosing loop.

cd [arg]
Change the current directory to arg. The shell parameter **$HOME** is the default arg.

eval [arg ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [arg ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input output arguments may appear and if no other arguments are given cause the shell input output to be modified.

exit [n]
Causes a non interactive shell to exit with the exit status specified by n. If n is omitted then the exit status is that of the last command executed. (An end of file will also exit from the shell.)

export [name ...]
The given names are marked for automatic export to the environment of subsequently-executed commands. If no arguments are given then a list of exportable names is printed.

login [arg ...]
Equivalent to 'exec login arg ...'.

newgrp [arg ...]
Equivalent to 'exec newgrp arg ...'.

read name ...
One line is read from the standard input; successive words of the input are assigned to the variables name in order, with leftover words to the last variable. The return code is 0 unless the end-of-file is encountered.

readonly [name ...]
The given names are marked readonly and the values of the these names may not be changed by subsequent assignment. If no arguments are given then a list of all readonly names is printed.

set [**-eknptuvx** [arg ...]]
-e If non interactive then exit immediately if a command fails.

-k All keyword arguments are placed in the environment for a command, not just those that precede the command name.
-n Read commands but do not execute them.
-t Exit after reading and executing one command.
-u Treat unset variables as an error when substituting.
-v Print shell input lines as they are read.
-x Print commands and their arguments as they are executed.
- Turn off the **-x** and **-v** options.

These flags can also be used upon invocation of the shell. The current set of flags may be found in **$-**.

Remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, etc. If no arguments are given then the values of all names are printed.

shift
The positional parameters from **$2...** are renamed **$1...**

times
Print the accumulated user and system times for processes run from the shell.

trap [arg] [n] ...
Arg is a command to be read and executed when the shell receives signal(s) n. (Note that arg is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. If arg is absent then all trap(s) n are reset to their original values. If arg is the null string then this signal is ignored by the shell and by invoked commands. If n is 0 then the command arg is executed on exit from the shell, otherwise upon receipt of signal n as numbered in signal(2). Trap with no arguments prints a list of commands associated with each signal number.

umask [nnn]
The user file creation mask is set to the octal value nnn (see umask(2)). If nnn is omitted, the current value of the mask is printed.

wait [n]
Wait for the specified process and report its termination status. If n is not given then all currently active child processes are waited for. The return code from this command is that of the process waited for.

Invocation.
If the first character of argument zero is -, commands are read from **$HOME/.profile**, if such a file exists. Commands

are then read as described below. The following flags are interpreted by the shell when it is invoked.

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

-s If the **-s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

-i If the **-i** flag is present or if the shell input and output are attached to a terminal (as told by gtty) then this shell is interactive. In this case the terminate signal SIGTERM (see signal(2)) is ignored (so that 'kill 0' does not kill an interactive shell) and the interrupt signal SIGINT is caught and ignored (so that **wait** is interruptable). In all cases SIGQUIT is ignored by the shell.

The remaining flags and arguments are described under the set command.

FILES

$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

test(1), exec(2),

DIAGNOSTICS

Errors detected by the shell, such as syntax errors cause the shell to return a non zero exit status. If the shell is being used non interactively then execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also **exit**).

NOTES

If << is used to provide standard input to an asynchronous process invoked by &, the shell gets mixed up about naming the input document. A garbage file /tmp/sh* is created, and the shell complains about not being able to find the file by another name.

NAME

shutdown - bring the system down gracefully.

SYNTAX

/etc/shutdown [time] [**su**]

DESCRIPTION

Shutdown will warn users that the system is about to go down, wait for them to log out, forcibly log out those who ignore the messages, kill any outstanding daemon and user processes and dismount any mounted filesystems.

Shutdown must be run from the console terminal by the super-user. If the time argument has not been given, the program will ask for the delay to allow users to log out, then send messages at one minute intervals encouraging them to log off. If all users log out before the interval ends, the program goes on to its next step.

Next, all processes except the init process and processes spawned from the console terminal are killed. To allow normal cleanup to take place, the executing processes are sent the SIGTERM signal followed by the SIGKILL signal after a short interval.

Next, all mounted filesystems are dismounted. For this step to be effective, the invoker's working directory should be on the root filesystem.

Finally, the system is set to single-user mode if the su argument has been given, otherwise the system is halted.

SEE ALSO

init(8), mount(1m)

NOTES

Shutdown's treatment of reluctant users is abrupt.

NAME

size - size of an object file

SYNTAX

size [object ...]

DESCRIPTION

For each object-file argument, size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in octal and decimal. Several object file formats are understood. If no file is specified, a.out is used.

If the file is an 8086 relocatable file with an x.out header, the text, data and bss sizes given are summaries of the contents of the relocation records.

SEE ALSO

a.out(5)

NAME

sleep - suspend execution for an interval

SYNTAX

sleep time

DESCRIPTION

Sleep suspends execution for time seconds. It is used to execute a command after a certain amount of time as in:

```
(sleep 105; command)&
```

or to execute a command every so often, as in:

```
while true
do
     command
     sleep 37
done
```

SEE ALSO

alarm(2), sleep(3)

NOTES

Time must be less than 65536 seconds.

NAME

sort - sort or merge files

SYNTAX

sort [-mubdfinrtx] [+pos1 [-pos2]] ... [-o name] [-T directory] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '-' means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b**.

r Reverse the sense of comparisons.

tx 'Tab character' separating fields is x.

The notation +pos1 -pos2 restricts a sort key to a field beginning at pos1 and ending just before pos2. Pos1 and pos2 each have the form m.n, optionally followed by one or more of the flags bdfinr, where m tells a number of fields to skip from the beginning of the line and n tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect n is counted from the first nonblank in the field; b is attached independently to pos2. A missing .n means .0; a missing -pos2 means the end of the line. Under the -tx option, fields are strings separated by x; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

m Merge only, the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

T The next argument is the name of a directory in which temporary files should be made.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

Examples. Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

```
sort -u +0f +0 list
```

Print the password file (passwd(5)) sorted by user id number (the 3rd colon-separated field).

```
sort -t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month day) entries. The options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
sort -um +0 -1 dates
```

FILES

/usr/tmp/stm*, /tmp/*: first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **-c**.

NOTES

Very long lines are silently truncated.

NAME

sp - convert narrow input to a wider format output.

SYNTAX

sp [width]

DESCRIPTION

sp Prints input in 8 character-wide columns onto the standard output. The optional argument, if numeric, specifies the width of the output.

Sp does not shorten long lines, it merely concatenates and spaces short ones.

EXAMPLE

ls | sp

SEE ALSO

prep (1), col (1)

NAME

spell, spellin, spellout - find spelling errors

SYNTAX

spell [option] ... [file] ...

/usr/src/cmd/spell/spellin [list]

/usr/src/cmd/spell/spellout [-d] list

DESCRIPTION

Spell collects words from the named documents, and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes or suffixes) from words in the spelling list are printed on the standard output. If no files are named, words are collected from the standard input.

Spell ignores most troff, tbl and eqn(1T) constructions.

Under the **-v** option, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated.

Under the -b option, British spelling is checked. Besides preferring centre, colour, speciality, travelled, etc., this option insists upon -ise in words like standardise.

Under the **-x** option, every plausible stem is printed with '=' for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective in respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy-y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by spell. Both expect a list of words, one per line, from the standard input. Spellin adds the words on the standard input to the preexisting list and places a new list on the standard output. If no list is specified, the new list is created from scratch. Spellout looks up each word in the standard input and prints on the standard output those that are missing from (or present on, with option -d) the hash list.

FILES

D=/usr/dict/hlist[ab]: hashed spelling lists, American & British
S=/usr/dict/hstop: hashed stop list
H=/usr/dict/spellhist: history file
/usr/lib/spell
deroff(1T), sort(1), tee(1), sed(1)

NOTES

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions.

NAME

spline - interpolate smooth curve

SYNTAX

spline [option] ...

DESCRIPTION

Spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output (R. W. Hamming, Numerical Methods for Scientists and Engineers, 2nd ed., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by graph(1).

The following options are recognized, each as a separate argument.

-a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

-k The constant k used in the boundary value computation

(2nd deriv. at end) = k*(2nd deriv. next to end)

is set by the next argument. By default k = 0.

-n Space output points so that approximately n intervals occur between the lower and upper x limits. (Default n = 100.)

-p Make output periodic, i.e. match derivatives at ends. First and last input values should normally agree.

-x Next 1 (or 2) arguments are lower (and upper) x limits. Normally these limits are calculated from the data. Automatic abcissas start at lower limit (default 0).

SEE ALSO

graph(1)

DIAGNOSTICS

When data is not strictly monotone in x, spline reproduces the input without interpolating extra points.

NOTES

A limit of 1000 input points is enforced silently. delim off

NAME

split - split a file into pieces

SYNTAX

split [-n] [file [name]]

DESCRIPTION

Split reads file and writes it in n-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is name with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if - is given in its stead, then the standard input file is used.

WARNING

1000 lines is usually less than 19 pages.
Lpr does not guarantee that it prints the files in the order given.

SEE ALSO

lpr (1), wc (1)

NAME

strings - find the printable strings in an object file

SYNTAX

strings [-] [**-o**] [-number] file ...

DESCRIPTION

Strings looks for ascii strings in a binary file. A string is any sequence of 4 or more printing characters ending with a newline or a null. Unless the - flag is given, strings only looks in the initialized data space of object files. If the **-o** flag is given, then each string is preceded by its offset in the file (in octal). If the -number flag is given then number is used as the minimum string length rather than 4.

Strings is useful for identifying random object files and many other things.

SEE ALSO

od(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

The algorithm for identifying strings is extremely primitive.

NAME

strip - remove selected parts of an object file

SYNTAX

strip [-dehrstx] file ...

DESCRIPTION

Strip is capable of removing most parts of an object file including the header, text, data, relocation records and the symbol table. Strip works directly upon the named files; nothing is written to the standard output.

This is useful to to remove an x.out header from an 8086 relocatable file, to save space by removing symbol table and relocation information when debugging is complete, or to provide a compact form of a namelist by removing the text and data.

Options and their meanings are:

-h Strip header and extended header.

-e Strip extended header.

-d Strip data and data relocation.

-t Strip text and text relocation.

-r Strip all relocation except x.out's "short form."

-x Strip all relocation.

-s Strip symbol table.

The effect of **strip -s** is the same as use of the **-s** option of ld. Strip without flags implies **-sr** .

FILES

/tmp/s* temporary files

SEE ALSO

ld(1S)

NAME

struct - structure Fortran programs

SYNTAX

struct [option] ... file

DESCRIPTION

Struct translates the Fortran program specified by file (standard input default) into a Ratfor program. Wherever possible, Ratfor control constructs replace the original Fortran. Statement numbers appear only where still necessary. Cosmetic changes are made, including changing Hollerith strings into quoted strings and relational operators into symbols (.e.g. '.GT.' into '>'). The output is appropriately indented.

The following options may occur in any order.

-s Input is accepted in standard format, i.e. comments are specified by a c, C, or * in column 1, and continuation lines are specified by a nonzero, nonblank character in column 6. Normally, a statement whose first nonblank character is not alphanumeric is treated as a continuation.

-i Do not turn computed goto statements into switches. (Ratfor does not turn switches back into computed goto statements.)

-a Turn sequences of else ifs into a non-Ratfor switch of the form

```
switch {
       case pred1: code
       case pred2: code
       case pred3: code
       default: code
      }
```

The case predicates are tested in order; the code appropriate to only one case is executed. This generalized form of switch statement does not occur in Ratfor.

-b Generate goto's instead of multilevel break statements.

-n Generate goto's instead of multilevel next statements.

-en If n is 0 (default), place code within a loop only if it can lead to an iteration of the loop. If n is nonzero, admit code segments with fewer than n

statements to a loop if otherwise the loop would have exits to several places including the segment, and the segment can be reached only from the loop.

FILES

/tmp/struct*
/usr/lib/struct/*

SEE ALSO

ratfor(1S)

NOTES

Struct knows Fortran 66 syntax, but not full Fortran 77 (alternate returns, IF...THEN...ELSE, etc.)
If an input Fortran program contains identifiers which are reserved words in Ratfor, the structured version of the program will not be a valid Ratfor program.
Extended range DO's generate cryptic errors.
Columns 73-80 are not special even when -s is in effect.
Will not generate Ratfor FOR statements.

NAME

stty - set terminal options

SYNTAX

stty [-l] [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output terminal. With no argument, it reports the current settings of the options. With only the -l argument it prints the current settings and also the tchar fields. The option strings are selected from the following set:

even	allow even parity
-even	disallow even parity
odd	allow odd parity
-odd	disallow odd parity
raw	raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
-raw	negate raw mode
cooked	same as '-raw'
cbreak	make each character available to read(2) as received; no erase and kill
-cbreak	make characters available to read only when newline is received
-nl	allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl	accept only new-line to end lines
echo	echo back every character typed
-echo	do not echo characters
lcase	map upper case to lower case
-lcase	do not map case
-tabs	replace tabs by spaces when printing
tabs	preserve tabs
ek	reset erase and kill characters back to normal # and @
erase c	set erase character to c. C can be of the form '^X' which is interpreted as a 'control X'.
kill c	set kill character to c. '^X' works here also.
cr0 cr1 cr2 cr3	select style of delay for carriage return (see ioctl(2))
nl0 nl1 nl2 nl3	select style of delay for linefeed
tab0 tab1 tab2 tab3	select style of delay for tab
ff0 ff1	select style of delay for form feed
bs0 bs1	select style of delay for backspace
tty33	set all modes suitable for the Teletype Corporation Model 33 terminal.
tty37	set all modes suitable for the Teletype Corporation Model 37 terminal.

vt05 set all modes suitable for Digital Equipment Corp. VT05 terminal

tn300 set all modes suitable for a General Electric TermiNet 300

ti700 set all modes suitable for Texas Instruments 700 series terminal

tek set all modes suitable for Tektronix 4014 terminal

hup hang up dataphone on last close.

-hup do not hang up dataphone on last close.

0 hang up phone line immediately

50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb
Set terminal baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

intrc _c_ set interrupt character to _c_. _C_ can be of the form '^_X_' which is interpreted as a 'control X'.

quitc _c_ set quit character to _c_. _C_ can be of the form '^_X_' which is interpreted as a 'control X'.

startc _c_
set start output character to _c_. _C_ can be of the form '^_X_' which is interpreted as a 'control X'.

stopc _c_ set stop output character to _c_. _C_ can be of the form '^_X_' which is interpreted as a 'control X'.

eofc _c_ set end-of-file character to _c_. _C_ can be of the form '^_X_' which is interpreted as a 'control X'.

brkc _c_ set input delimiter (like nl) character to _c_. _C_ can be of the form '^_X_' which is interpreted as a 'control X'. The strings 'del', 'DEL', 'OFF', and 'off' may be used where '^X' is used to specify the delete character or to disable the function.

SEE ALSO
ioctl(2), tabs(1)

NAME

su - become super-user or another user

SYNTAX

su [-] [name [arg ...]]

DESCRIPTION

Su allows one to become another user without logging off. The default user name is root (i.e., super-user).

To use su, the appropriate password must be supplied (unless one is already super-user). If the password is correct, su will execute a new shell with the user ID set to that of the specified user. To restore normal user ID privileges, type an **EOF** to the new shell.

Any additional arguments are passed to the shell, permitting the super-user to run shell procedures with restricted privileges (an arg of the form -c string executes string via the shell). When additional arguments are passed, /bin/sh is always used. When no additional arguments are passed, su uses the shell specified in the password file.

An initial - flag causes the environment to be changed to the one that would be expected if the user actually logged in again. This is done by invoking the shell with an arg0 of -su causing the .profile in the home directory of the new user ID to be executed. Otherwise, the environment is passed along with the possible exception of **$PATH**, which is set to **:/bin:/etc:/usr/bin** for root. Note that the .profile can check arg0 for -sh or -su to determine how it was invoked.

FILES

/etc/passwd	system's password file
$HOME/.profile	user's profile

SEE ALSO

env(1), login(1), sh(1), environ(7).

NAME

sum - sum and count blocks in a file

SYNTAX

sum file

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. It is typically used to look for bad spots, or to validate a file communicated over some transmission line.

SEE ALSO

wc(1)

DIAGNOSTICS

'Read error' is indistinguishable from end of file on most devices; check the block count.

NAME

sync - update the super block

SYNTAX

sync

DESCRIPTION

Sync executes the sync system primitive. If the system is to be stopped, sync must be called to insure file system integrity. See sync(2) for details.

SEE ALSO

sync(2), update(8)

NAME

sysadmin

SYNTAX

/etc/sysadmin

DESCRIPTION

Sysadmin is a script for performing file system backups and for restoring files from backup disks. It can do a daily incremental backup (level 9), or a periodic full backup (level 0). It can provide a listing of the files backed up and also has a facility to restore individual files from a backup.

Sysadmin operates on XENIX format diskettes, either single or double sided. The version provided backs up the root file system. The script can be edited to operate on additional file systems if required.

You must be super-user to use this program.

FILES

/tmp/backup.list

SEE ALSO

dump(1), restor(1), mkfs(1), dumpdir(1).

NAME

tabs - set terminal tabs

SYNTAX

tabs [-n] [terminal]

DESCRIPTION

Tabs sets the tabs on a variety of terminals. Various of the terminal names given in term(7) are recognized; the default is, however, suitable for most 300 baud terminals. If the -n flag is present then the left margin is not indented as is normal.

SEE ALSO

stty(1), term(7)

NAME

tail - deliver the last part of a file

SYNTAX

tail ±number[lbc] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +number from the beginning, or -number from the end of the input. Number is counted in units of lines, blocks or characters, according to the appended option **l**, **b** or **c**. When no units are specified, counting is by lines.

SEE ALSO

dd(1)

NOTES

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tar - tape archiver

SYNTAX

tar [key] [name ...]

DESCRIPTION

Tar saves and restores files on magtape or other storage device. Its actions are controlled by the key argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r The named files are written on the end of the tape. The c function implies this.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, the entire content of the tape is extracted. Note that if multiple entries specifying the same file are on the tape, the last one overwrites all earlier.

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

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

c Create a new tape; writing begins on the beginning of the tape instead of after the last file. This command implies r.

The following characters may be used in addition to the letter which selects the function desired.

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

v Normally tar does its work silently. The **v** (verbose) option causes it to type the name of each

file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.

w causes tar to print the action to be taken followed by file name, then wait for user confirmation. If a word beginning with 'y' is given, the action is performed. Any other input means don't do it.

f causes tar to use the next argument as the name of the archive instead of /dev/mt?. If the name of the file is '-', tar writes to standard output or reads from standard input, whichever is appropriate. Thus, tar can be used as the head or tail of a filter chain Tar can also be used to move hierarchies with the command

```
cd fromdir; tar cf - . | (cd todir; tar xf -)
```

b causes tar to use the next argument n as the blocking factor for tape records. On devices other than magnetic tape, this causes tar to buffer I/O using an n block buffer. (Tar blocks are 512 bytes each.) The default is 1, the maximum is 20. This option should only be used with raw magnetic tape or floppy disk archives (See **f** above). The record size is determined automatically when reading tapes (key letters 'x' and 't').

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

m tells tar to not restore the modification times. The mod time will be the time of extraction.

k causes tar to use the next argument as the size of an archive volume in kilobytes. The minimum value allowed is 250. This option is useful when the archive is not intended for a magnetic tape device, but for some fixed size device, such as floppy disk (See **f** above). Very large files are split into "extents" across volumes. When restoring from a multi-volume archive, tar only prompts for a new volume if a split file has been partially restored.

n indicates the archive device is not a magnetic tape. The **k** option implies this. Listing and extracting the contents of an archive are sped because tar can seek over files it wishes to skip. Sizes are printed in Kilobytes instead of tape

blocks.

FILES

/dev/mt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if enough memory is not available to hold the link tables.

EXAMPLES

To backup a disk directory tree to tape using raw I/O and a blocking factor of 20:

```
tar cfb /dev/rmt1 20 directory_name
```

To restore the above files from tape to disk:

```
tar xf /dev/rmt1 directory_name
```

SEE ALSO

dump(1), restor(1), copy(1), dd(1), physio(5)

NOTES

There is no way to ask for the n-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow.
In order to update (**r** or **u** option) a tar archive one must not use raw magtape and not use the **b** option. This applies both when updating and when the archive was first created.
The current limit on file name length is 100 characters.
Systems with a 1K-byte file system (as this one does) cannot specify raw disk devices unless the **b** option is used to specify an even number of blocks. This means that one cannot update (**r** or **u** option) a raw-mode disk partition.

NAME

tbl - format tables for nroff or troff

SYNTAX

tbl [files] ...

DESCRIPTION

Tbl is a preprocessor for formatting tables for nroff(1T) or troff(1T). The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables and reformatted. Details are given in the reference manual.

As an example, letting \t represent a tab (which should be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\tl644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields

Household Population		
Town	Households	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Branchburg	1644	3.49
Bridgewater	7897	3.81
Far Hills	240	3.19

If no arguments are given, tbl reads the standard input, so it may be used as a filter. When it is used with eqn or neqn the tbl command should be first, to minimize the volume of data passed through pipes.

SEE ALSO

nroff(1T), troff(1T), eqn(1T)
M. E. Lesk, TBL.

NAME

tee - pipe fitting

SYNTAX

tee [-i] [-a] [file] ...

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the files. Option **-i** ignores interrupts; option **-a** causes the output to be appended to the files rather than overwriting them.

NAME

test - condition command

SYNTAX

test expr

DESCRIPTION

test evaluates the expression expr, and if its value is true then returns zero exit status; otherwise, a non zero exit status is returned. test returns a non zero exit if there are no arguments.

The following primitives are used to construct expr.

-r file true if the file exists and is readable.

-w file true if the file exists and is writable.

-f file true if the file exists and is not a directory.

-d file true if the file exists and is a directory.

-s file true if the file exists and has a size greater than zero.

-t [fildes]
true if the open file whose file descriptor number is fildes (1 by default) is associated with a terminal device.

-z s1 true if the length of string s1 is zero.

-n s1 true if the length of the string s1 is nonzero.

s1 = s2 true if the strings s1 and s2 are equal.

s1 **!=** s2 true if the strings s1 and s2 are not equal.

s1 true if s1 is not the null string.

n1 **-eq** n2
true if the integers n1 and n2 are algebraically equal. Any of the comparisons **-ne**, **-gt**, **-ge**, **-lt**, or **-le** may be used in place of **-eq**.

These primaries may be combined with the following operators:

! unary negation operator

-a binary and operator

-o binary or operator

(expr)
parentheses for grouping.

-a has higher precedence than **-o**. Notice that all the operators and flags are separate arguments to test. Notice also that parentheses are meaningful to the Shell and must be escaped.

SEE ALSO
sh(1), find(1)

NAME

time - time a command

SYNTAX

time command

DESCRIPTION

The given command is executed; after it is complete, time prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on the diagnostic output stream.

NOTES

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

touch - update date last modified of a file

SYNTAX

touch [-c] file ...

DESCRIPTION

Touch attempts to set the modified date of each file. This is done by reading a character from the file and writing it back.

If a file does not exist, an attempt will be made to create it unless the **-c** option is specified.

NAME

tr - translate characters

SYNTAX

tr [-cds] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in string1 are mapped into the corresponding characters of string2. When string2 is short it is padded to the length of string1 by duplicating its last character. Any combination of the options -cds may be used: -c complements the set of characters in string1 with respect to the universe of characters whose ASCII codes are 01 through 0377 octal; -d deletes all input characters in string1; -s squeezes all strings of repeated output characters that are in string2 to single characters.

In either string the notation a-b means a range of characters from a to b in increasing ASCII order. The character '\' followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. A '\' followed by any other character stands for that character.

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect '\' from the Shell. 012 is the ASCII code for newline.

```
tr -cs A-Za-z '\012' <file1 >file2
```

SEE ALSO

ed(1), ascii(7)

NOTES

Won't handle ASCII NUL in string1 or string2; always deletes NUL from input.

NAME

troff - text typesetting

SYNTAX

troff [option] ... [file] ...

DESCRIPTION

Troff formats text in the named files. Troff is part of the nroff/troff family of text formatters. Nroff is used to format files for output to a lineprinter or daisy wheel printer; troff to a Graphic Systems C/A/T phototypesetter.

If no file argument is present, the standard input is read. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input. The options, which may appear in any order so long as they appear before the files, are:

-olist Print only pages whose page numbers appear in the comma-separated list of numbers and ranges. A range N-M means pages N through M; an initial -N means from the beginning to page N; and a final N- means from N to the end.

-nN Number first generated page N.

-sN Stop every N pages. Troff stops the phototypesetter every N pages, produces a trailer to allow changing cassettes, and then resumes when the typesetter's start button is pressed.

-mname Prepend the macro file /usr/lib/tmac/tmac.name to the input files.

-raN Set register a (one-character) to N.

-i Read standard input after the input files are exhausted.

-q Invoke the simultaneous input-output mode of the .rd request.

-t Direct output to the standard output instead of the phototypesetter.

-f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-w Wait until phototypesetter is available, if currently busy.

-b Report whether the phototypesetter is busy or

available. No text processing is done.

-a Send a printable ASCII approximation of the results to the standard output.

-pN Print all characters in point size N while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

If the file /usr/adm/tracct is writable, troff keeps phototypesetter accounting records there. The integrity of that file may be secured by making troff a 'set user-id' program.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/font/*	font width tables for troff
/dev/cat	phototypesetter
/usr/adm/tracct	accounting statistics for /dev/cat

SEE ALSO

J. F. Ossanna, Nroff/Troff user's manual
B. W. Kernighan, A TROFF Tutorial
eqn(1T), nroff(1T), tbl(1T)

NAME

true - return true

SYNTAX

true

DESCRIPTION

True does nothing except return with a zero exit value. False(1), true's counterpart does nothing except return with a non-zero exit value. True is typically used in shell procedures such as:

```
while true
do
     command
done
```

SEE ALSO

sh(1), false (1)

DIAGNOSTICS

True has exit status zero.

NAME

tset - set terminal modes

SYNTAX

tset [-] [**-hrsuIQS**] [-e[c]] [-E[c]] [**-k**[c]] [**-m** [ident][test baudrate]:type] [type]

DESCRIPTION

Tset causes terminal dependent processing such as setting erase and kill characters, setting or resetting delays, and the like. It is driven by the /etc/ttytype and /etc/termcap files.

The type of terminal is specified by the type argument. The type may be any type given in /etc/termcap. If type is not specified, the terminal type is read from /etc/htmp (the home directory and terminal type database), or the environment **TERM**, unless the **-h** flag is set or any **-m argument was given.** In this case the type is read from /etc/ttytype (the port name to terminal type database). The port name is determined by a ttyname(3) call on the diagnostic output. If the port is not found in /etc/ttytype the terminal type is set to unknown.

Ports for which the terminal type is indeterminate are identified in /etc/ttytype as dialup, plugboard, etc. The user can specify how these identifiers should map to an actual terminal type. The mapping flag, **-m**, is followed by the appropriate identifier (a 4 character or longer substring is adequate), an optional test for baud rate, and the terminal type to be used if the mapping conditions are satisfied. If more than one mapping is specified, the first correct mapping prevails. A missing identifier matches all identifiers. Baud rates are specified as with stty(1), and are compared with the speed of the diagnostic output. The test may be any combination of: **>**, **=**, **<**, **@**, and **!**. (Note: **@** is a synonym for **=** and **!** inverts the sense of the test. Remember to escape characters meaningful to the shell.)

If the type as determined above begins with a question mark, the user is asked if s/he really wants that type. A null response means to use that type; otherwise, another type can be entered which will be used instead. (The question mark must be escaped to prevent filename expansion by the shell.)

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is the default erase character ('#' on standard systems), the erase character is changed to a Control-H (backspace). The -e flag sets the erase character to be the named character c on all terminals, so to override this option one can say **-e#**. The default for c is the backspace character on the terminal,

usually Control-H. The -E flag is identical to -e except that it only operates on terminals that can backspace; it might be used if you had the misfortune to be stuck with an ASR33. The -k option works similarly, with c defaulting to Control-X. No kill processing is done if -k is not specified. In all of these flags, '^X' where X is any character is equivalent to control-X.

On version 6 systems, the terminal type specified in htmp is updated unless -u is specified.

The - option prints the terminal type on the standard output; this can be used to get the terminal type by saying:

```
set termtype = `tset -`
```

If no other options are given, tset operates in "fast mode" and only outputs the terminal type, bypassing all other processing. The -s option outputs "setenv" commands (if your default shell is csh) or "export" and assignment commands (if your default shell is the Bourne shell); the -S option only outputs the strings to be placed in the environment variables. The -s option can be used as:

```
`tset -s ...`
```

Actually, this is not possible because of a problem in the shell. Instead, if you are using the Bourne shell, use:

```
tset -s ... > /tmp/tset$$
/tmp/tset$$
rm /tmp/tset$$
```

If you are using csh, use:

```
set noglob
set term=('tset -S ....')
setenv TERM $term[1]
setenv TERMCAP "$term[2]"
unset term
unset noglob
```

The -r option prints the terminal type on the diagnostic output. The -Q option supresses printing the "Erase set to" and "Kill set to" messages. The -I option supresses outputing the terminal initialization strings.

Tset is most useful when included in the .login (for csh(1)) or .profile (for sh(1)) file executed automatically at login, with -m mapping used to specify the terminal type you most frequently dial in on.

EXAMPLES

```
tset gt42
tset -mdialup\>300:adm3a -mdialup:dw2 -Qr -e#
tset -m dial:ti733 -m plug:\?hp2621 -m unknown:\? -e -k^U
```

FILES

/etc/htmp Terminal type database (version 6 only)
/etc/ttytype Port name to terminal type map database
/etc/termcap Terminal capability database

SEE ALSO

setenv(1), ttys(5), termcap(5), stty(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

For compatibility with earlier versions of tset, the following flags are accepted and mapped internally as shown:

```
-d type   ->   -m dialup:type
-p type   ->   -m plugboard:type
-a type   ->   -m arpanet:type
```

These flags will disappear eventually.

NAME

tsort - topological sort

SYNTAX

tsort [file]

DESCRIPTION

Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input file. If no file is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1S)

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

NOTES

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty - get terminal name

SYNTAX

tty

DESCRIPTION

Tty prints the pathname of the user's terminal.

DIAGNOSTICS

'not a tty' if the standard input file is not a terminal.

NAME

umount - dismount file system

SYNTAX

/etc/umount special

DESCRIPTION

Umount announces to the system that the removable file system previously mounted on device special is to be removed. First, any pending I/O for the file system is completed, and the file system is flagged clean. For a full explanation of the mounting process see mount(1).

FILES

/etc/mtab: mount table

SEE ALSO

mount(1), mount(2), mtab(5)

NAME

uniq - report repeated lines in a file

SYNTAX

uniq [-udc [+n] [-n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. Note that repeated lines must be adjacent in order to be found; see sort(1). If the -u flag is used, just the lines that are not repeated in the original file are output. The -d option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the -u and -d mode outputs.

The -c option supersedes -u and -d and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The n arguments specify skipping an initial portion of each line in the comparison:

-n The first n fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+n The first n characters are ignored. Fields are skipped before characters.

SEE ALSO

sort(1), comm(1)

NAME

units - conversion program

SYNTAX

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
     * 2.54000e+00
     / 3.93701e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have: 15 pounds force/in2
You want: atm
     * 1.02069e+00
     / 9.79730e-01
```

Units only does multiplicative scale changes. Thus it can convert Kelvin to Rankine, but not Centigrade to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

```
pi    ratio of circumference to diameter
c     speed of light
e     charge on an electron
g     acceleration of gravity
force      same as g
mole Avogadro's number
water      pressure head per unit height of water
au    astronomical unit
```

'Pound' is a unit of mass. Compound names are run together, e.g. 'lightyear'. British units that differ from their US counterparts are prefixed thus: 'brgallon'. Currency is denoted 'belgiumfranc', 'britainpound', ...

For a complete list of units, 'cat /usr/lib/units'.

FILES

/usr/lib/units

NOTES

Currency conversions are not current.

NAME

uucp, uulog - unix to unix copy

SYNTAX

uucp [option] ... source-file ... destination-file

uulog [option] ...

DESCRIPTION

Uucp copies files named by the source-file arguments to the destination-file argument. A file name may be a path name on your machine, or may have the form

system-name!pathname

where 'system-name' is taken from a list of system names which uucp knows about. Shell metacharacters ?*[] appearing in the pathname part will be expanded on the appropriate system.

Pathnames may be one of

(1) a full pathname;

(2) a pathname preceded by ~user; where user is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

If the result is an erroneous pathname for the remote system the copy will fail. If the destination-file is a directory, the last part of the source-file name is used.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see chmod(2)).

The following options are interpreted by uucp.

-d Make all necessary directories for the file copy.

-c Use the source file when copying out rather than copying the file to the spool directory.

-m Send mail to the requester when the copy is complete.

Uulog maintains a summary log of uucp and uux(1) transactions in the file '/usr/spool/uucp/LOGFILE' by gathering information from partial log files named '/usr/spool/uucp/LOG.*.?'. It removes the partial log files.

The options cause uulog to print logging information:

-ssys
Print information about work involving system sys.

-uuser
Print information about work done for the specified user.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/* - other data and program files

SEE ALSO

uux(1), mail(1)
D. A. Nowitz, Uucp Implementation Description

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will very likely not be able to fetch files by pathname; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary pathnames.

NOTES

All files received by uucp will be owned by uucp.
The -m option will only work sending files or receiving a single file. (Receiving multiple files specified by special shell characters ?*[] will not activate the -m option.)

NAME

uux - unix to unix command execution

SYNTAX

uux [-] command-string

DESCRIPTION

Uux will gather 0 or more files from various systems, execute a command on a specified system and send standard output to a file on a specified system.

The command-string is made up of one or more arguments that look like a shell command line, except that the command and file names may be prefixed by system-name!. A null system-name is interpreted as the local system.

File names may be one of

(1) a full pathname;

(2) a pathname preceded by ~xxx; where xxx is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

The '-' option will cause the standard input to the uux command to be the standard input to the command-string.

For example, the command

```
uux "!diff usg!/usr/dan/f1 pwba!/a4/dan/f1 > !fi.diff"
```

will get the f1 files from the usg and pwba machines, execute a diff command and put the results in f1.diff in the local directory.

Any special shell characters such as <>;| should be quoted either by quoting the entire command-string, or quoting the special characters as individual arguments.

FILES

/usr/uucp/spool - spool directory
/usr/uucp/* - other data and programs

SEE ALSO

uucp(1)
D. A. Nowitz, Uucp implementation description

WARNING

An installation may, and for security reasons generally will, limit the list of commands executable on behalf of an

incoming request from uux. Typically, a restricted site will permit little other than the receipt of mail via uux.

NOTES

Only the first command of a shell pipeline may have a system-name!. All other commands are executed on the system of the first command.
The use of the shell metacharacter * will probably not do what you want it to do.
The shell tokens << and >> are not implemented.
There is no notification of denial of execution on the remote machine.

NAME

vi - screen oriented (visual) display editor based on ex

SYNTAX

vi [**-t** tag] [**-r**] [**+**lineno] name ...

DESCRIPTION

Vi (visual) is a screen oriented text editor based on **ex**(1). Ex and vi run the same code; it is possible to get to the command mode of ex from within vi and vice-versa. See ex(1) for a list of the available ex commands that can be executed as colon commands when in vi.

FILES

SEE ALSO

ex(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

Scans with / and ? begin on the next line, skipping the remainder of the current line.

Software tabs using **^T** work only immediately after the autoindent.

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The wrapmargin option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Occasionally inverse video scrolls up into the file from a diagnostic on the last line.

Saving text on deletes in the named buffers is inefficient.

The source command does not work when executed as **:source**; there is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a : escape. To use these on a **:global** you must **Q** to ex command mode, execute them, and then reenter the screen editor with vi or open.

NAME

wait - await completion of process

SYNTAX

wait

DESCRIPTION

Wait until all processes started with & have completed, and report on abnormal terminations.

Because the wait(2) system call must be executed in the parent process, the Shell itself executes wait, without creating a new process.

SEE ALSO

sh(1)

NOTES

Not all the processes of a 3- or more-stage pipeline are children of the Shell, and thus can't be waited for.

NAME

wall - write to all users

SYNTAX

/etc/wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message, preceded by 'Broadcast Message ...', to all logged in users.

The sender should be super-user to override any protections the users may have invoked.

FILES

/dev/tty?
/etc/utmp

SEE ALSO

mesg(1), write(1)

DIAGNOSTICS

'Cannot send to ...' when the open on a user's tty file fails.

NAME

wc - word count

SYNTAX

wc [-lwc] [name ...]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of characters delimited by spaces, tabs or newlines.

If the optional argument is present, just the specified counts (lines, words or characters) are selected by the letters **l**, **w**, or **c**.

NAME

who - who is on the system

SYNTAX

who [who-file] [am I]

DESCRIPTION

Who, without an argument, lists the login name, terminal name, and login time for each current XENIX user.

Without an argument, who examines the /etc/utmp file to obtain its information. If a file is given, that file is examined. Typically the given file will be /usr/adm/wtmp, which contains a record of all the logins since it was created. Then who lists logins, logouts, and crashes since the creation of the wtmp file. Each login is listed with user name, terminal name (with '/dev/' suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with 'x' in the place of the device name, and a fossil time indicative of when the system went down.

With two arguments, as in 'who am I' (and also 'who are you'), who tells who you are logged in as.

FILES

/etc/utmp

SEE ALSO

getuid(2), utmp(5)

NAME

write - write to another user

SYNTAX

write user [ttyname]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message

Message from yourname yourttyname...

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal or an interrupt is sent. At that point write writes 'EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the ttyname argument may be used to indicate the appropriate terminal name.

Permission to write may be denied or granted by use of the mesg command. At the outset writing is allowed. Certain commands, in particular nroff and pr(1) disallow messages in order to prevent messy output.

If the character '!' is found at the beginning of a line, write calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using write: when you first write to another user, wait for him to write back before starting to send. Each party should end each message with a distinctive signal-(o) for 'over' is conventional-that the other may reply. (oo) for 'over and out' is suggested when conversation is about to be terminated.

FILES

/etc/utmp to find user
/bin/sh to execute '!'

SEE ALSO

mesg(1), who(1), mail(1)

NAME

xstr - extract strings from C programs to implement shared strings

SYNTAX

xstr [-c] [-] [file]

DESCRIPTION

Xstr maintains a file strings into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

xstr -c name

will extract the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An approporiate declaration of xstr is prepended to the file. The resulting C text is placed in the file x.c, to then be compiled. The strings from this file are placed in the strings data base if they are not there already. Repeated strings and strings which are suffices of existing strings do not cause changes to the data base.

After all components of a large program have been compiled a file xs.c declaring the common xstr space can be created by a command of the form

xstr

This xs.c file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. A command

xstr name

creates files x.c and xs.c as before, without using or affecting any strings file in the same directory.

It may be useful to run xstr after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. Xstr reads from its standard input when the argument '-' is given. An appropriate command sequence for running xstr after the C preprocessor is:

```
cc -E name.c | xstr -c -
cc -c x.c
mv x.o name.o
```

Xstr does not touch the file strings unless new items are added, thus make can avoid remaking xs.o unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array 'xstr'
/tmp/xs*	Temp file when 'xstr name' doesn't touch strings

SEE ALSO

mkstr(1S)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

If a string is a suffix of another string in the data base, but the shorter string is seen first by xstr both strings will be placed in the data base, when just placing the longer one there will do.

NAME

yacc - yet another compiler-compiler

SYNTAX

yacc [**-vd**] grammar

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, y.tab.c, must be compiled by the C compiler to produce a program yyparse. This program must be loaded with the lexical analyzer program, yylex, as well as main and yyerror, an error handling routine. These routines must be supplied by the user; Lex(1S) is useful for creating lexical analyzers usable by yacc.

If the **-v** flag is given, the file y.output is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the **-d** flag is used, the file y.tab.h is generated with the define statements that associate the yacc-assigned 'token codes' with the user-declared 'token names'. This allows source files other than y.tab.c to access the token codes.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs
/lib/liby.a	library with default 'main' and 'yyerror'

SEE ALSO

lex(1S)
LR Parsing by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974.
YACC - Yet Another Compiler Compiler by S. C. Johnson.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the y.output file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

NOTES

Because file names are fixed, at most one yacc process can

be active in a given directory at a time.

NAME

yes - be infinitely affirmative

SYNTAX

yes [arg]

DESCRIPTION

Yes repeatedly outputs y, or if a single argument is given, then arg is output repeatedly. The command will continue indefinitely unless aborted. Useful in pipes to commands that prompt for input and require a 'y' response for a yes. In this case, yes terminates when the command it pipes to terminates, so that no infinite loop occurs.

NAME

intro, errno - introduction to system calls and error numbers

SYNTAX

#include <errno.h>

DESCRIPTION

Section 2 of this manual lists all the entries into the XENIX system kernel. The pages for this section are provided only as part of the XENIX Software Development Package. Most of these calls have an error return. An error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details. An error number is also made available in the external variable errno. Errno is set on erroneous calls; its value is undefined on successful calls.

There is a table of messages associated with each error, and a routine for printing the message; See perror(3). The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in <errno.h>, and the messages available using perror.

0 Error 0
Unused.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
The process whose number was given to signal and ptrace does not exist, or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

SYSTEM CALLS

INTRO(2)

5 EIO I/O error
Some physical I/O error occurred during a read or write. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice that does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on line in or no disk is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5120 bytes is presented to exec.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see a.out(5).

9 EBADF Bad file number
Either a file descriptor refers to no open file, a read request is made to a file that is open only for writing, or a write request is made to a file that is open only for reading.

10 ECHILD No children
A wait was executed by a process that has no existing or unwaited-for children.

11 EAGAIN No more processes
A fork failed because the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough core
During an exec or break, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g. in mount.

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment).

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g. link.

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to chdir.

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in signal, reading or writing a file for which seek has generated a negative pointer. Also set by math functions, see intro(3).

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more opens can be accepted.

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in stty or gtty is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program that is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about 10.0^9 bytes).

28 ENOSPC No space left on device
During a write to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An lseek was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 links to a file.

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

35 EUCLEAN Structure needs cleaning
An attempt was made to mount(2) a file system whose superblock is not flagged clean().

36 EDEADLOCK Would deadlock
A process' attempt to lock a file region would cause a deadlock between processes vying for control of that region.

37 ENOTNAM Not a semaphore
A creatsem(), opensem(), waitsem(), or sigsem() was issued using an invalid semaphore identifier.

38 ENAVAIL Not available
An opensem(), waitsem() or sigsem() was issued to a semaphore that has not been initialized by a call to creatsem().

A sigsem() was issued to a semaphore out of sequence; i.e., before the process has issued the corresponding waitsem() to the semaphore.

An nbwaitsem() was issued to a semaphore guarding a resource that is currently in use by another process.

The semaphore on which a process was waiting has been left in an inconsistent state when the process controlling the semaphore exits without relinquishing control properly; i.e., without issuing a waitsem() on the semaphore.

SEE ALSO

intro(3)

NAME

access - determine accessibility of file

SYNTAX

```
access(name, mode)
char *name;
```

DESCRIPTION

Access checks the given file name for accessibility according to mode, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if name cannot be found or if any of the desired access modes would not be granted. On disallowed accesses -1 is returned and the error code is in errno. 0 is returned from successful tests.

Access uses the real user ID in place of the effective user ID and the real group ID in place of the effective group ID, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by access, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but exec will fail unless it is in proper format.

SEE ALSO

stat(2)

NAME

acct - turn accounting on or off

SYNTAX

```
acct(file)
char *file;
```

DESCRIPTION

The system is prepared to write a record in an accounting file for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to file. An argument of 0 causes accounting to be turned off.

The accounting file format is given in acct(5).

SEE ALSO

acct(5), sa(1)

DIAGNOSTICS

On error, -1 is returned and errno is set to indicate the The file must exist and the call may be exercised only by the super-user. It is erroneous to try to turn on accounting when it is already on.

NOTES

No accounting is produced for programs running when a crash occurs. In particular nonterminating programs are never accounted for.

NAME

alarm - schedule signal after specified time

SYNTAX

alarm(seconds)
unsigned seconds;

DESCRIPTION

Alarm causes signal SIGALRM, see signal(2), to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is cancelled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), signal(2), sleep(3)

NAME

brk, sbrk, break - change core allocation

SYNTAX

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Brk sets the system's idea of the lowest location not used by the program (called the break) to addr (rounded up to the next unit of memory management resolution, usually either 64 or 128 bytes.) Locations not less than addr and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function sbrk, incr more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via exec the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use break.

SEE ALSO

exec(2), malloc(3), end(3)

DIAGNOSTICS

Zero is returned if the break could be set; -1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break.

NOTES

Setting the break in the range 0177701 to 0177777 on some systems is the same as setting it to zero.

NAME

chdir, chroot - change default directory

SYNTAX

```
chdir(dirname)
char *dirname;

chroot(dirname)
char *dirname;
```

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. Chdir causes this directory to become the current working directory, the starting point for path names not beginning with '/'.

Chroot sets the root directory, the starting point for path names beginning with '/'. The call is restricted to the super-user.

SEE ALSO

cd(1)

DIAGNOSTICS

Zero is returned if the directory is changed; -1 is returned if the given name is not that of a directory or is not searchable.

NAME

chmod - change mode of file

SYNTAX

chmod(name, mode)
char *name;

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by name has its mode changed to mode. Modes are constructed by ORing together some combination of the following:

```
04000 set user ID on execution
02000 set group ID on execution
01000 save text image after execution
00400 read by owner
00200 write by owner
00100 execute (or search if a directory) by owner
00070 read, write, execute (search) by group
00007 read, write, execute (search) by others
```

If an executable file is set up for sharing (**-n** or **-i** option of ld(1)) then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worth while for heavily used commands.

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

SEE ALSO

chmod(1)

DIAGNOSTIC

Zero is returned if the mode is changed; -1 is returned if name cannot be found or if current user is neither the owner of the file nor the super-user.

NAME

chown - change owner and group of a file

SYNTAX

```
chown(name, owner, group)
char *name;
```

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by name has its owner and group changed as specified. Only the super-user may execute this call.

SEE ALSO

chown(1), passwd(5)

DIAGNOSTICS

Zero is returned if the owner is changed; -1 is returned on illegal owner changes.

NAME

close - close a file

SYNTAX

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an open, creat, dup or pipe(2) call, close closes the associated file. A close of all files is automatic on exit, but since there is a limit on the number of open files per process, close is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain file descriptors may be closed by exec(2) (see ioctl(2)).

SEE ALSO

creat(2), open(2), pipe(2), exec(2), ioctl(2)

DIAGNOSTICS

Zero is returned if a file is closed; -1 is returned for an unknown file descriptor.

NAME

creat - create a new file

SYNTAX

```
creat(name, mode)
char *name;
```

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called name, given as the address of a null-terminated string. If the file did not exist, it is given mode mode, as modified by the process's mode mask (see umask(2)). Also see chmod(2) for the construction of the mode argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

The mode given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a creat, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write(2), close(2), chmod(2), umask (2)

DIAGNOSTICS

The value -1 is returned if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

NAME

creatsem - create an instance of a binary semaphore

SYNTAX

```
sem_num = creatsem(sem_name,mode);
int sem_num,mode
char *sem_name;
```

DESCRIPTION

Creatsem defines a binary semaphore named by sem name (given as the address of a null-terminated string of less than 15 characters) to be used by waitsem(2) and sigsem(2) to manage mutually exclusive access to a resource, shared variable, or critical section of a program. Creatsem returns a unique semaphore number sem num which is then used as the parameter in waitsem and sigsem calls. Semaphores are special files of 0 length. The file name space is used to provide unique identifiers for semaphores. Mode sets the accessability of the semaphore using the same format as file access bits. Access to a semaphore is granted only on the basis of the read access bit: the write and execute bits are ignored.

A semaphore can be operated on only by a synchronizing primitive (waitsem or sigsem), or by a function that initializes it to some value (creatsem), or by a function that opens the semaphore for use by a process (opensem). Synchronizing primitives are guaranteed to be executed without interruption once started. These primitives are used by associating a semaphore with each resource (including critical code sections) to be protected.

The process controlling the semaphore should issue

```
sem_num = creatsem("semaphore", mode);
```

to create, initialize, and open the semaphore for that process. All other processes using the semaphore should issue

```
sem_num = opensem("semaphore")
```

to access the semaphore's identification value. NOTE that a process cannot open and use a semaphore that has not been initialized by a call to creatsem (errno = ENAVAIL), nor should a process open a semaphore more than once in one period of execution. It is expected that a semaphore be reinitialized using creatsem at the start of each active use of the semaphore; the semaphore can be assumed to hold valid information only as long as it is opened and in use by active processes. Both the creating and opening processes use

```
waitsem(sem_num) and sigsem(sem_num)
```

to use the semaphore sem num.

SEE ALSO

opensem(2x), waitsem(2x), sigsem(2x)

DIAGNOSTICS

Creatsem returns the value (int) -1 if an error occurs: if the semaphore named by sem_name is already open for use by other processes (errno = EEXIST) or if the file specified exists but is not a semaphore type (errno = ENOTNAM).

NAME

dup, dup2 - duplicate an open file descriptor

SYNTAX

```
dup(fildes)
int fildes;

dup2(fildes, fildes2)
int fildes, fildes2;
```

DESCRIPTION

Given a file descriptor returned from an open, pipe, or creat call, dup allocates another file descriptor synonymous with the original. The new file descriptor is returned.

In the second form of the call, fildes is a file descriptor referring to an open file, and fildes2 is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). Dup2 causes fildes2 to refer to the same file as fildes. If fildes2 already referred to an open file, it is closed first.

SEE ALSO

creat(2), open(2), close(2), pipe(2)

DIAGNOSTICS

The value -1 is returned if: the given file descriptor is invalid; there are already too many open files.

NAME

execl, execv, execle, execve, execlp, execvp, exec, exece, environ - execute a file

SYNTAX

```
execl(name, arg0, argl, ..., argn, 0)
char *name, *arg0, *argl, ..., *argn;

execv(name, argv)
char *name, *argv[ ];

execle(name, arg0, argl, ..., argn, 0, envp)
char *name, *arg0, *argl, ..., *argn, *envp[ ];

execve(name, argv, envp);
char *name, *argv[ ], *envp[ ];

extern char **environ;
```

DESCRIPTION

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

Files remain open across exec unless explicit arrangement has been made; see ioctl(2). Ignored signals remain ignored across these calls, but signals that are caught (see signal(2)) are reset to their default values.

Each user has a real user ID and group ID and an effective user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. Exec changes the effective user and group ID to the owner of the executed file if the file has the 'set-user-ID' or 'set-group-ID' modes. The real user ID is not affected.

The name argument is a pointer to the name of the file to be executed. The pointers arg[0], arg[1] ... address null-terminated strings. Conventionally arg[0] is the name of the file.

From C, two interfaces are available. Execl is useful when a known file with known arguments is being called; the arguments to execl are the character strings constituting the file and the arguments. The first argument is conventionally the same as the filename (or its last component). A 0 argument must end the argument list.

The execv version is useful when the number of arguments is unknown in advance. The arguments to execv are the name of

the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. As indicated, argc is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is directly usable in another execv because argv[argc] is 0.

Envp is a pointer to an array of strings that constitute the environment of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell sh(1) passes an environment entry for each global shell variable defined when the program is called. See environ(5) for some conventionally used names. The C run-time start-off routine places a copy of envp in the global cell environ, which is used by execv and execl to pass the environment to any subprograms executed by the current program. The exec routines use lower-level routines as follows to pass an environment explicitly:

```
execle(file, arg0, argl, . . . , argn, 0, environ);
execve(file, argv, environ);
```

Execlp and execvp are called with the same arguments as execl and execv, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

FILES

/bin/sh shell, invoked if command file found by execlp or execvp

SEE ALSO

fork(2), environ(5)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see a.out(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is -1. Even for the super-user, at least one of the

execute-permission bits must be set for a file to be executed.

NOTES

If execvp is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of argv[0] and argv[-1] will be modified before return.

NAME

exit - terminate process

SYNTAX

```
exit(status)
int status;

_exit(status)
int status;
```

DESCRIPTION

Exit is the normal means of terminating a process. Exit closes all the process's files and notifies the parent process if it is executing a wait. The low-order 8 bits of status are available to the parent process.

This call can never return.

The C function exit may cause cleanup actions before executing the final machine system exit instruction. The function _exit circumvents all cleanup.

SEE ALSO

wait(2)

NAME

fork - create a new process

SYNTAX

fork( )

DESCRIPTION

Fork is the only way new processes are created. The new process's core image is a copy of that of the caller of fork. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. The process ID is used by wait(2).

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

SEE ALSO

wait(2), exec(2)

DIAGNOSTICS

Returns -1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.

NAME

getpid - get process identification

SYNTAX

getpid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

SEE ALSO

mktemp(3)

NAME

getuid, getgid, geteuid, getegid - get user and group identity

SYNTAX

getuid( )

geteuid( )

getgid( )

getegid( )

DESCRIPTION

Getuid returns the real user ID of the current process, geteuid the effective user ID. The real user ID identifies the person who is logged in, in contrast to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the 'set user ID' mode, to find out who invoked them.

Getgid returns the real group ID, getegid the effective group ID.

SEE ALSO

setuid(2)

NAME

indir - indirect system call

SYNTAX

sys indir; call

DESCRIPTION

The system call at the location call is executed. Execution resumes after the indir call. This provides an indirect means of making a system call. It is obsolete but provided for compatibility.

NAME

ioctl, stty, gtty - control device

SYNTAX

```
#include <sgtty.h>

ioctl(fildes, request, argp)
struct sgttyb *argp;

stty(fildes, argp)
struct sgttyb *argp;

gtty(fildes, argp)
struct sgttyb *argp;
```

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). See section 4 for a discussion of the available devices.

For certain status setting and status inquiries about terminal devices, the functions stty and gtty are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
```

respectively; see tty(4).

The following two calls, however, apply to any open file:

```
ioctl(fildes, FIOCLEX, NULL);
ioctl(fildes, FIONCLEX, NULL);
```

The first causes the file to be closed automatically during a successful exec operation; the second reverses the effect of the first.

SEE ALSO

stty(1), tty(4), exec(2)

DIAGNOSTICS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended.

NOTES

Strictly speaking, since ioctl may be extended in different ways to devices with different properties, argp should have an open-ended declaration like

```
union { struct sgttyb ...; ... } *argp;
```

The important thing is that the size is fixed by 'struct sgttyb'.

NAME

kill - send signal to a process

SYNTAX

kill(pid, sig);

DESCRIPTION

Kill sends the signal sig to the process specified by the process number pid . See signal(2) for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see tty(4).

If the process number is -1, and the user is the super-user, the signal is broadcast universally except to processes 0 and 1, the scheduler and initialization processes, see init(8).

Processes may send signals to themselves.

SEE ALSO

signal(2), kill(1)

DIAGNOSTICS

Zero is returned if the process is killed; -1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

NAME

link - link to a file

SYNTAX

```
link(name1, name2)
char *name1, *name2;
```

DESCRIPTION

A link to name1 is created; the link has the name name2. Either name may be an arbitrary path name.

SEE ALSO

ln(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; -1 is returned when name1 cannot be found; when name2 already exists; when the directory of name2 cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when a file has too many links.

NAME

lock - lock a process in primary memory

SYNTAX

lock(flag)

DESCRIPTION

If the flag argument is non-zero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is unlocked. This call may only be executed by the super-user.

NOTES

Locked processes interfere with the compaction of primary memory and can cause deadlock. This system call is not considered a permanent part of the system.

NAME

locking - lock or unlock a file region for reading or writing

SYNTAX

```
locking(fildes, mode, size);
int fildes, mode;
long size;
```

DESCRIPTION

Locking allows a specified number of bytes in a file to be controlled by the locking process. Other processes which attempt to read or write a portion of the file containing the locked region may sleep until the area becomes unlocked depending upon the mode in which the file region was locked. A process that attempts to write to or read a file region that has been locked against reading and writing by another process (using the LK_LOCK or LK_NBLCK mode) will sleep until the region of the file has been released by the locking process. A process that attempts to write to a file region that has been locked against writing by another process (using the LK_RLCK or LK_NBRLCK mode) will sleep until the region of the file has been released by the locking process, but a read request for that file region will proceed normally.

A process that attempts to lock a region of a file that contains areas that have been locked by other processes will sleep if it has specified the LK_LOCK or LK_RLCK mode in its lock request, but will return with the error EACCESS if it specified LK_NBLCK or LK_NBRLCK.

Fildes is the value returned from a successful creat, open, dup, or pipe system call.

Mode specifies the type of lock operation to be performed on the file region. The available values for mode are:

LK_UNLCK 0
: Unlock the specified region. The calling process releases a region of the file it had previously locked.

LK_LOCK 1
: Lock the specified region. The calling process will sleep until the entire region is available if any part of it has been locked by a different process. The region is then locked for the calling process and no other process may read or write in any part of the locked region. (lock against read and write).

LK_NBLCK 2

LOCKING(2X)

Lock the specified region. If any part of the region is already locked by a different process, return the error EACCESS instead of waiting for the region to become available for locking (non-blocking lockrequest).

LK_RLCK 3
Same as LK_LOCK except that the locked region may be read by other processes (read permitted lock).

LK_NBRLCK 4
Same as LK_NBLCK except that the locked region may be read by other processes (non-blocking, read permitted lock).

Size is the number of contiguous bytes to be locked or unlocked. The region to be locked starts at the current offset in the file. If size is 0, the entire file (up to a maximum of 2^30 bytes) is locked or unlocked. Size may extend beyond the end of the file, in which case only the process issuing the lock call may access or add information to the file within the boundary defined by size.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another process' locked area. Thus calls to locking, read, or write scan for a deadlock prior to sleeping on a locked region. An error return is made if sleeping on the locked region would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs, or when adjacent regions are locked, the regions are combined into a single area if the mode of the lock is the same (i.e.; either read permitted or regular lock). If the mode of the overlapping locks differ, the locked areas will be assigned assuming that the most recent request must be satisfied. Thus if a read only lock is applied to a region, or part of a region, that had been previously locked by the same process against both reading and writing, the area of the file specified by the new lock will be locked for read only, while the remaining region, if any, will remain locked against reading and writing. There is no arbitrary limit to the number of regions which may be locked in a file. There is however a system-wide limit on the total number of locked regions. This limit is 200 for XENIX systems.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional locked element to hold the separated

section. If the lock table is full, an error is returned, and the requested region is not released. Only the process which locked the file region may unlock it. An unlock request for a region that the process does not have locked, or that is already unlocked, has no effect. When a process terminates, all locked regions controlled by that process are unlocked.

If a process has done more than one open on a file, all locks put on the file by that process will be released on the first close of the file.

Although no error is returned if locks are applied to special files or pipes, read/write operations on these types of files will ignore the locks. Locks may not be applied to a directory.

SEE ALSO

creat(2), open(2), read(2), write(2), dup(2), close(2), lseek(2)

DIAGNOSTICS

Locking returns the value (int) -1 if an error occurs: if any portion of the region has been locked by another process for the LK_LOCK & LK_RLCK actions and the lock request is to test only (errno = EACCESS), if the file specified is a directory (errno = EACCESS), if locking the region would cause a deadlock (errno = EDEADLOCK), or if there are no more free internal locks (errno = EDEADLOCK).

NAME

lseek, tell - move read/write pointer

SYNTAX

```
long lseek(fildes, offset, whence)
long offset;

long tell(fildes)
```

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read or write pointer for the file is set as follows:

If whence is 0, the pointer is set to offset bytes.

If whence is 1, the pointer is set to its current location plus offset.

If whence is 2, the pointer is set to the size of the file plus offset.

The returned value is the resulting pointer location.

The obsolete function tell(fildes) is identical to lseek(fildes, 0L, 1).

Seeking far beyond the end of a file, then writing, creates a gap or 'hole', which occupies no physical space and reads as zeros.

SEE ALSO

open(2), creat(2), fseek(3)

DIAGNOSTICS

-1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

NOTES

Lseek is a no-op on character special files.

NAME

mknod - make a directory or a special file

SYNTAX

```
mknod(name, mode, addr)
char *name;
```

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by name. The mode of the new file (including directory and special file bits) is initialized from mode. (The protection part of the mode is modified by the process's mode mask; see umask(2)). The first block pointer of the i-node is initialized from addr. For ordinary files and directories addr is normally zero. In the case of a special file, addr specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(1), mknod(1), filsys(5)

DIAGNOSTICS

Zero is returned if the file has been made; -1 if the file already exists or if the user is not the super-user.

NAME

mount, umount - mount or remove file system

SYNTAX

```
mount(special, name, rwflag)
char *special, *name;

umount(special)
char *special;
```

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file special; from now on, references to file name will refer to the root file on the newly mounted file system. Special and name are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. Name must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

The rwflag argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the special file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation. Any pending I/O for the file system is completed, and the file system is marked clean.

SEE ALSO

mount(1), intro(2), mknod(1M), fsck(1M)

DIAGNOSTICS

Mount returns 0 if the action occurred; -1 if special is inaccessible or not an appropriate file; if name does not exist; if special is already mounted; if name is in use; or if there are already too many file systems mounted. If the file system was unclean, 'errno' == EUCLEAN. Use fsck(1m) to clean the file system.

Umount returns 0 if the action occurred; -1 if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

NAME

nice - set program priority

SYNTAX

nice(incr)

DESCRIPTION

The scheduling priority of the process is augmented by incr. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without inconveniencing other users.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range -20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by fork(2). For a privileged process to return to normal priority from an unknown state, nice should be called successively with arguments -40 (goes to priority -20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO

nice(1)

NAME

open - open file for reading or writing

SYNTAX

```
#include <fcntl.h>
int open(path,oflag)
char *path;
int oflag;
```

DESCRIPTION

Path points to a path name naming a file. Open opens a descriptor for the named file and sets the file status flags according to the value of oflag. Oflag values are constructed by OR-ing flags from the following list (only one of the first three flags may be used):

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading or writing
O_WSYNC	on write, wait for I/O to complete

DIAGNOSTICS

No error is returned if oflag is incorrect. If any valid bits in the flag parameter are set, open ignores the invalid ones. If the flag is completely spurious - if no correct bits are set - the mode is set to 0, an undefined mode.

NAME

opensem - opens a semaphore

SYNTAX

```
sem_num = opensem(sem_name);
int sem_num;
char *sem_name;
```

DESCRIPTION

Opensem opens a semaphore named by sem_name and returns the unique semaphore identification number sem_num used by waitsem and sigsem. Creatsem should always be called to initialize and reset the semaphore before any process attempts to open the semaphore.

SEE ALSO

creatsem(2x), waitsem(2x), sigsem(2x)

DIAGNOSTICS

Opensem returns the value (int) -1 if an error occurs: if the semaphore named does not exist (errno = ENOENT), if the file specified is not a semaphore file; i. e., a file previously created (by some process) using a call to creatsem (errno = ENOTNAM), or if the file specified is not currently open due to a call to creatsem (errno = ENAVAIL).

NAME

pause - stop until signal

SYNTAX

pause( )

DESCRIPTION

Pause never returns normally. It is used to give up control while waiting for a signal from kill(2) or alarm(2).

SEE ALSO

kill(1), kill(2), alarm(2), signal(2), setjmp(3)

NAME

phys - allow a process to access physical addresses

SYNTAX

phys(segreg, size, physadr)

DESCRIPTION

The argument segreg specifies a process virtual (data-space) address range of 8K bytes starting at virtual address segregx8K bytes. This address range is mapped into physical address physadrx64 bytes. Only the first sizex64 bytes of this mapping is addressable. If size is zero, any previous mapping of this virtual address range is nullified. For example, the call

```
phys(6, 1, 0177775);
```

will map virtual addresses 0160000-0160077 into physical addresses 017777500-017777577. In particular, virtual address 0160060 is the PDP-11 console located at physical address 017777560.

This call may only be executed by the super-user.

DIAGNOSTICS

The function value zero is returned if the physical mapping is in effect. The value -1 is returned if not super-user, if segreg is not in the range 0-7, if size is not in the range 0-127, or if the specified segreg is already used for other than a previous call to phys.

NOTES

This system call is obviously very machine dependent and very dangerous. This system call is not considered a permanent part of the system.

NAME

pipe - create an interprocess channel

SYNTAX

```
pipe(fildes)
int fildes[2];
```

DESCRIPTION

The pipe system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor fildes[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor fildes[0] will pick up the data. Writes with a count of 4096 bytes or less are atomic; no other process can intersperse data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent fork calls) will pass data through the pipe with read and write calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

SEE ALSO

sh(1), read(2), write(2), fork(2)

DIAGNOSTICS

The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

NOTES

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

NAME

profil - execution time profile

SYNTAX

```
profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;
```

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by bufsiz. After this call, the user's program counter (pc) is examined each clock tick (60th second); offset is subtracted from it, and the result multiplied by scale. If the resulting number corresponds to a word inside buff, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777(8) gives a 1-1 mapping of pc's to words in buff; 077777(8) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of buff (producing a non-interrupting core clock).

Profiling is turned off by giving a scale of 0 or 1. It is rendered ineffective by giving a bufsiz of 0. Profiling is turned off when an exec is executed, but remains on in child and parent both after a fork. Profiling may be turned off if an update in buff would cause a memory fault.

SEE ALSO

monitor(3), prof(1)

NOTES

You should use the monitor(3) call. The prof(1) program may require the buffer size to be equal to or smaller than the program size.

NAME

ptrace - process trace

SYNTAX

```
#include <signal.h>

ptrace(request, pid, addr, data)
int *addr;
```

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a request argument. Generally, pid is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt.' See signal(2) for the list. Then the traced process enters a stopped state and its parent is notified via wait(2). When the child is in the stopped state, its core image can be examined and modified using ptrace. If desired, another ptrace request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the request argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at addr is returned. If I and D space are separated, request 1 indicates I space, 2 D space. Addr must be even. The child must be stopped. The input data is ignored.

3 The word of the system's per-process data area corresponding to Addr is returned. Addr must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the user structure in the system.

4,5 The given data is written at the word in the process's address space corresponding to addr, which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space.

Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The data argument is taken as a signal number and the child's execution continues at location addr as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If addr is (int *)1 then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. This is part of the mechanism for implementing breakpoints. On other processors, appropriate machine-dependent strategies are used.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The wait call is used to determine when a process stops; in such a case the 'termination' status returned by wait has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, ptrace inhibits the set-user-id facility on subsequent exec(2) calls. If a traced process calls exec, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

SEE ALSO

wait(2), signal(2), adb(1)

DIAGNOSTICS

The value -1 is returned if request is invalid, pid is not a traceable process, addr is out of bounds, or data specifies an illegal signal number.

NOTES

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use 'illegal instruction' signals at a very high rate) could be efficiently debugged.

The error indication, -1, is a legitimate function value; check intro(2) and errno to determine the meaning of a particular error number.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

Differing processors and configurations will necessarily create some differences in functionality.

NAME

rdchk - check if there is data to be read

SYNTAX

```
rdchk(fdes);
int fdes;
```

DESCRIPTION

Rdchk checks to see if a process will block if it attempts to read the file designated by fdes. Rdchk returns 1 if there is data to be read or if it is the end of the file (EOF). In this context, the proper sequence of calls using rdchk is:

```
if(rdchk(fildes) > 0) read(fildes, buffer, nbytes);
```

SEE ALSO

read(2)

DIAGNOSTICS

Rdchk returns -1 if an error occurs (e.g., EBADF), 0 if the process will block if it issues a read and 1 if it is okay to read. EBADF is returned if a rdchk is done on a semaphore file or if the file specified doesn't exist.

NAME

read - read from file

SYNTAX

read(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful open, creat, dup, or pipe call. Buffer is the location of nbytes contiguous bytes into which the input will be placed. It is not guaranteed that all nbytes bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open(2), creat(2), dup(2), pipe(2)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is -1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous nbytes, file descriptor not that of an input file.

NAME

setuid, setgid - set user and group ID

SYNTAX

setuid(uid)

setgid(gid)

DESCRIPTION

The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user, unless the argument is the real ID.

SEE ALSO

getuid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; -1 is returned otherwise.

NAME

shutdn - flush block I/O and halt CPU

SYNTAX

```
#include <sys/filsys.h>

shutdn (sblk, ~sblk)
struct filsys *sblk;
```

DESCRIPTION

Shutdn causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O. The super-blocks of all writable file systems are flagged 'clean', so that they can be remounted without cleaning when XENIX is rebooted. Shutdn then prints "Normal System Shutdown" on the console and halts the cpu.

If sblk is non-zero, it specifies the address of a super block which will be written to the root device as the last I/O before the halt. This facility is provided to allow file system repair programs to supercede the system's copy of the root super block with one of their own. The address of the new super block is boolean inverted to form the second argument. This is an attempt to reduce the liklihood of a program accidentally invoking shutdn and destroying the root file system.

Shutdn locks out all other processes while it is doing its work. However, it is recommended that user processes be killed off (see kill(1)) before calling shutdn as some types of disk activity could cause file systems to not be flagged 'clean'.

The caller must be the super-user.

SEE ALSO

fsck(1m), haltsys(8), shutdown(8), mount(2)

NAME

signal - catch or ignore signals

SYNTAX

```
#include <signal.h>

(*signal(sig, func))()
(*func)();
```

DESCRIPTION

A signal is generated by some abnormal event, initiated either by user at a typewriter (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but a signal call allows them either to be ignored or to cause an interrupt to a specified location. The list of signals follows:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	illegal instruction trap
SIGEMT	7*	emulator instruction trap
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGPIPE	13	write on a pipe or link with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

The starred signals in the list above cause a core image if not caught or ignored.

If func is SIG_DFL, the default action for signal sig is reinstated; this default is termination, sometimes with a core image. If func is SIG_IGN the signal is ignored. Otherwise when the signal occurs func will be called with the signal number as argument. A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another signal call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a read or write(2) on a slow device (like a typewriter; but not a file); and during pause or wait(2). When such a signal occurs, the saved user status is arranged in

such a way that when return from the signal-catching takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of signal is the previous (or initial) value of func for the particular signal.

After a fork(2) the child inherits all signals. Exec(2) resets all caught signals to default action.

SEE ALSO

kill(1), kill(2), ptrace(2), setjmp(3)

DIAGNOSTICS

The value (int)-1 is returned if the given signal is out of range.

NOTES

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its func argument are problematical.

NAME

sigsem - signal a process waiting on a semaphore

SYNTAX

```
sigsem(sem_num);
int sem_num;
```

DESCRIPTION

Sigsem signals a process that is waiting on the semaphore sem num that it may proceed and use the resource governed by the semaphore. Sigsem is used in conjunction with waitsem(2x) to allow synchronization of processes wishing to access a resource. One or more processes may waitsem on the given semaphore and will be put to sleep until the process which currently has access to the resource issues a sigsem call. If there are any waiting processes, sigsem causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

SEE ALSO

creatsem(2x), opensem(2x), waitsem(2x)

DIAGNOSTICS

Sigsem returns the value (int) -1 if an error occurs: if sem num does not refer to a semaphore type file (errno = ENOTNAM), if sem num has not been previously opened by opensem (errno = EBADF), or if the process issuing a sigsem call is not the current "owner" of the semaphore; i. e., if the process has not issued a waitsem call before the sigsem (errno = ENAVAIL).

NAME

stat, fstat - get file status

SYNTAX

```
#include <sys/types.h>
#include <sys/stat.h>

stat(name, buf)
char *name;
struct stat *buf;

fstat(fildes, buf)
struct stat *buf;
```

DESCRIPTION

Stat obtains detailed information about a named file. Fstat obtains the same information about an open file known by the file descriptor from a successful open, creat, dup or pipe(2) call.

Name points to a null-terminated string naming a file; buf is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable. The layout of the structure pointed to by buf as defined in <stat.h> is given below. St mode is encoded according to the '#define' statements.

```
struct  stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        unsigned short st_mode;
        short   st_nlink;
        short   st_uid;
        short   st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};

#define S_IFMT  0170000 /* type of file */
#define S_IFDIR 0040000 /* directory */
#define S_IFCHR 0020000 /* character special */
#define S_IFBLK 0060000 /* block special */
#define S_IFREG 0100000 /* regular */
#define S_IFMPC 0030000 /* multiplexed char special */
#define S_IFMPB 0070000 /* multiplexed block special */
#define S_IFNAM 0050000 /* semaphore, message system special */
```

```
#define S_ISUID 0004000 /* set user id on execution */
#define S_ISGID 0002000 /* set group id on execution */
#define S_ISVTX 0001000 /* save swapped text even after use */
#define S_IREAD 0000400 /* read permission, owner */
#define S_IWRITE 0000200 /* write permission, owner */
#define S_IEXEC 0000100 /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see chmod(2)). The defined types, ino t, off t, time t, name various width integer values; dev t encodes major and minor device numbers; their exact definitions are in the include file <sys/types.h> (see types(5).

When fildes is associated with a pipe, fstat reports an ordinary file with restricted permissions. The size is the number of bytes queued in the pipe.

st atime is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. st mtime is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. st ctime is set both both by writing and changing the i-node.

SEE ALSO

ls(1), filsys(5)

DIAGNOSTICS

Zero is returned if a status is available; -1 if the file cannot be found.

NAME

stime - set time

SYNTAX

```
stime(tp)
long *tp;
```

DESCRIPTION

Stime sets the system's idea of the time and date. Time, pointed to by tp, is measured in seconds from 0000 GMT Jan 1, 1970. Only the super-user may use this call.

SEE ALSO

date(1), time(2), ctime(3)

DIAGNOSTICS

Zero is returned if the time was set; -1 if user is not the super-user.

NAME

sync - update super-block

SYNTAX

sync( )

DESCRIPTION

Sync causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified inodes, and delayed block I/O.

It should be used by programs which examine a file system, for example icheck, df, etc. It is mandatory before a boot.

SEE ALSO

sync(1), update(8)

NOTES

The writing, although scheduled, is not necessarily complete upon return from sync.

NAME

time, ftime - get date and time

SYNTAX

```
long time(0)

long time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If tloc is nonnull, the return value is also stored in the place to which tloc points.

The ftime entry fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
        time_t  time;
        unsigned short millitm;
        short   timezone;
        short   dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

date(1), stime(2), ctime(3)

NAME

times - get process times

SYNTAX

```
times(buffer)
struct tbuffer *buffer;
```

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ=60 in North America.

After the call, the buffer will appear as follows:

```
struct tbuffer {
     long proc_user_time;
     long proc_system_time;
     long child_user_time;
     long child_system_time;
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time(1), time(2)

NAME

umask - set file creation mode mask

SYNTAX

umask(complmode)

DESCRIPTION

Umask sets a mask used whenever a file is created by creat(2) or mknod(2): the actual mode (see chmod(2)) of the newly-created file is the logical **and** of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 0 (no restrictions). The mask is inherited by child processes.

SEE ALSO

creat(2), mknod(2), chmod(2)

NAME

unlink - remove directory entry

SYNTAX

```
unlink(name)
char *name;
```

DESCRIPTION

Name points to a null-terminated string. Unlink removes the entry for the file pointed to by name from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO

rm(1), link(2)

DIAGNOSTICS

Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

NAME

utime - set file times

SYNTAX

```
#include <sys/types.h>
utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

The utime call uses the 'accessed' and 'updated' times in that order from the timep vector to set the corresponding recorded times for file.

The caller must be the owner of the file or the super-user. The 'inode-changed' time of the file is set to the current time.

SEE ALSO

stat (2)

NAME

wait - wait for process to terminate

SYNTAX

```
wait(status)
int *status;

wait(0)
```

DESCRIPTION

Wait causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last wait, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of -1 returned). The normal return yields the process ID of the terminated child. In the case of several children several wait calls are needed to learn of all the deaths.

If (int)status is nonzero, the high byte of the word pointed to receives the low byte of the argument of exit when the child terminated. The low byte receives the termination status of the process. See signal(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See ptrace(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit(2), fork(2), signal(2)

DIAGNOSTICS

Returns -1 if there are no children not previously waited for.

NAME

waitsem - await access to a resource governed by a semaphore
nbwaitsem - check access to a resource governed by a semaphore

SYNTAX

```
waitsem(sem_num);
int sem_num;

nbwaitsem(sem_num);
int sem_num;
```

DESCRIPTION

Waitsem gives the calling process access to the resource governed by the semaphore sem num. If the resource is in use by another process, waitsem will put the process to sleep until the resource becomes available; nbwaitsem will return the error ENAVAIL. Waitsem and nbwaitsem are used in conjunction with sigsem to allow synchronization of processes wishing to access a resource. One or more processes may waitsem on the given semaphore and will be put to sleep until the process which currently has access to the resource issues sigsem. Sigsem causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

SEE ALSO

creatsem(2x), opensem(2x), sigsem(2x)

DIAGNOSTICS

Waitsem returns the value (int) -1 if an error occurs: if sem num has not been previously opened by a call to opensem or creatsem (errno = EBADF), or if sem num does not refer to a semaphore type file (errno = ENOTNAM). All processes waiting (or attempting to wait) on the semaphore when the process controlling the semaphore exits without relinquishing control (thereby leaving the resource in an undeterminate state) return with errno set to ENAVAIL.

NAME

write - write on a file

SYNTAX

write(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful open, creat, dup, or pipe(2) call.

Buffer is the address of nbytes contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

If the flag O_WSYNC was set when the file was opened, the data will be written before the write operation returns to the process rather than buffered internally for later output. Setting O_WSYNC with the open system call guarantees the consistency of a file as soon as the write call returns even if the machine crashes. Since this will cause significant extra I/O, system and user throughput will be decreased.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat(2), open(2), pipe(2),fcntl(7)

DIAGNOSTICS

Returns -1 on error: bad descriptor, buffer address, or count; physical I/O errors.

NAME

intro - introduction to library functions

SYNTAX

```
#include <stdio.h>

#include <math.h>
```

DESCRIPTION

This section describes functions that may be found in various libraries. It does not describe functions that directly invoke XENIX system primitives: these are described in section 2. The pages for this section are provided only as part of the XENIX Software Development Package. Functions are divided into various libraries distinguished by the section number at the top of the page:

(3) These functions, together with those of section 2 and those marked (3S), constitute library libc, which is automatically loaded by the C compiler cc(1). The link editor ld(1) searches this library under the '-lc' option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3M) These functions constitute the math library, libm. The link editor searches this library under the '-lm' option. Declarations for these functions may be obtained from the include file <math.h>.

(3S) These functions constitute the 'standard I/O package', see stdio(3). These functions are in the library libc already mentioned. Declarations for these functions may be obtained from the include file <stdio.h>.

(3X) Various specialized libraries have not been given distinctive captions. The files in which these libraries are found are named on the appropriate pages.

FILES

/lib/libc.a
/lib/libm.a, /usr/lib/libm.a (one or the other)

SEE ALSO

stdio(3), nm(1S), ld(1S), cc(1S), intro(2)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable errno (see intro(2)) is set to the value EDOM or ERANGE. The values of EDOM and ERANGE are

defined in the include file <math.h>.

NAME

abort - generate I/O trap fault

SYNTAX

abort()

DESCRIPTION

Abort causes a signal that normally terminates the process with a core dump, which may be used for debugging.

SEE ALSO

adb(1), signal(2), exit(2)

DIAGNOSTICS

Usually 'core dumped' from the shell.

NAME

abs - integer absolute value

SYNTAX

abs(i)

DESCRIPTION

Abs returns the absolute value of its integer operand.

SEE ALSO

floor(3) for fabs

NOTES

You get what the hardware gives on the largest negative integer.

NAME

assert - program verification

SYNTAX

#include <assert.h>

assert (expression)

DESCRIPTION

Assert is a macro that indicates expression is expected to be true at this point in the program. It causes an exit(2) with a diagnostic comment on the standard output when expression is false (0). Compiling with the cc(1) option **-DNDEBUG** effectively deletes assert from the program.

DIAGNOSTICS

'Assertion failed: file f line n.' F is the source file and n the source line number of the assert statement.

NAME

atof, atoi, atol - convert ASCII to numbers

SYNTAX

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION

These functions convert a string pointed to by nptr to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

Atoi and atol recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3)

NOTES

There are no provisions for overflow.

NAME

crypt, setkey, encrypt - DES encryption

SYNTAX

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
```

DESCRIPTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to crypt is a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The salt string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The other entries provide (rather primitive) access to the actual DES algorithm. The argument of setkey is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the encrypt entry is likewise a character array of length 64 containing 0's and 1's. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by setkey. If edflag is 0, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

passwd(1), passwd(5), login(1), getpass(3)

NOTES

The return value points to static data whose content is overwritten by each call.

NAME

ctime, localtime, gmtime, asctime, timezone - convert date and time to ASCII

SYNTAX

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

DESCRIPTION

Ctime converts a time pointed to by clock such as returned by time(2) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

Localtime and gmtime return pointers to structures containing the broken-down time. Localtime corrects for the time zone and possible daylight savings time; gmtime converts directly to GMT, which is the time XENIX uses. Asctime converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm { /* see ctime(3) */
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday =

0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the standard U.S.A. daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

Timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g. in Afghanistan timezone(-(60*4+30), 0) is appropriate because it is 4:30 ahead of GMT and the string **GMT+4:30** is produced.

SEE ALSO

time(2)

NOTES

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii - character classification
toupper, tolower, toascii - character transformation

SYNTAX

#include <ctype.h>

isalpha(c)

. . .

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. Isascii is defined on all integer values; the rest are defined only where isascii is true and on the single non-ASCII value EOF (see stdio(3)).

isalpha	c is a letter
isupper	c is an upper case letter
islower	c is a lower case letter
isdigit	c is a digit
isxdigit	c is a hexadecimal digit
isalnum	c is an alphanumeric character
isspace	c is a space, tab, carriage return, newline, or formfeed
ispunct	c is a punctuation character (neither control nor alphanumeric)
isprint	c is a printing character, code 040(8) (space) through 0176 (tilde)
iscntrl	c is a delete character (0177) or ordinary control character (less than 040).
isascii	c is an ASCII character, code less than 0200

The macros below transform ASCII-coded integer values and non-ascii characters in a repeatable way.

toupper
: c transforms lower case letters to upper case, but is undefined for other values.

tolower
: c transforms upper case letters to lower case, but is undefined for other values.

toascii
: c transforms a non-ascii character to the corresponding ascii character, without disturbing ascii characters. Since EOF is not a character, mapping to ascii has undesirable properties.

SEE ALSO

ascii(7), stdio(3), getc(3), putc(3)

NAME

curses - screen functions with optimal cursor motion

SYNTAX

cc [flags] files -lcurses -ltermlib [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the refresh() tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine initscr() must be called before any of the other routines that deal with windows and screens are used.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold,
termcap (5), stty (2), setenv (3), setenv (3),

FUNCTIONS

addch(ch)	Add a character to stdscr
addstr(str)	Add a string to stdscr
box(win,vert,hor)	Draw a box around a window
crmode()	Set cbreak mode
clear()	Clear stdscr
clearok(scr,boolf)	Set clear flag for scr
clrtobot()	Clear to bottom on stdscr
clrtoeol()	Clear to end of line on stdscr
delwin(win)	Delete win
echo()	Set echo mode
erase()	Erase stdscr
getch()	Get a char through stdscr
getstr(str)	Get a string through stdscr
gettmode()	Get tty modes
getyx(win,y,x)	Get (y,x) co-ordinates
inch()	Get char at current (y,x) co-ordinates
initscr()	Initialize screens
leaveok(win,boolf)	Set leave flag for win
longname(termbuf,name)	Get long name from termbuf
move(y,x)	Move to (y,x) on stdscr
mvcur(lasty,lastx,newy,newx)	Actually move cursor
newwin(lines,cols,begin_y,begin_x)	Create a new window
nl()	Set newline mapping
nocrmode()	Unset cbreak mode
noecho()	Unset echo mode
nonl()	Unset newline mapping
noraw()	Unset raw mode

```
overlay(win1,win2)  Overlay win1 on win2
overwrite(win1,win2)
                    Overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)
                    Printf on stdscr
raw()               Set raw mode
refresh()           Make current screen look like stdscr
restty()            Reset tty flags to stored value
savetty()           Stored current tty flags
scanw(fmt,arg1,arg2,...)
                    Scanf through stdscr
scroll(win)         Scroll win one line
scrollok(win,boolf)
                    Set scroll flag
setterm(name)       Set term variables for name
unctrl(ch)          Printable version of ch
waddch(win,ch)      Add char to win
waddstr(win,str)    Add string to win
wclear(win)         Clear win
wclrtobot(win)      Clear to bottom of win
wclrtoeol(win)      Clear to end of line on win
werase(win)         Erase win
wgetch(win)         Get a char through win
wgetstr(win,str)    Get a string through win
winch(win)          Get char at current (y,x) in win
wmove(win,y,x)      Set current (y,x) co-ordinates on win
wprintw(win,fmt,arg1,arg2,...)
                    Printf on win
wrefresh(win)       Make screen look like win
wscanw(win,fmt,arg1,arg2,...)
                    Scanf through win
```

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NAME

dbminit, fetch, store, delete, firstkey, nextkey - data base subroutines

SYNTAX

```
typedef struct { char *dptr; int dsize; } datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two filesystem accesses. The functions are obtained with the loader option **-ldbm**.

Keys and contents are described by the datum typedef. A datum specifies a string of dsize bytes pointed to by dptr. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by dbminit. At the time of this call, the files **file.dir** and **file.pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by fetch and data is placed under a key by store. A key (and its associated contents) is deleted by delete. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of firstkey and nextkey. Firstkey will return the first key in the database. With any key nextkey will return the next key in the database. This code will traverse the data base:

```
for(key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

DIAGNOSTICS

All functions that return an int indicate errors with negative values. A zero return indicates ok. Routines that return a datum indicate errors with a null (0) dptr.

NOTES

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older XENIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 512 bytes). Moreover all key/content pairs that hash together must fit on a single block. Store will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by firstkey and nextkey depends on a hashing function, not on anything interesting.

NAME

defopen, defread - read default entries

SYNTAX

```
int defopen(filename)
char *filename;

char *defread(pattern)
char *pattern;
```

DESCRIPTION

Defopen and defread are a pair of routines designed to allow easy access to default definition files. **XENIX** is normally distributed in binary form; the use of default files allows OEMS or site administrators to customize utility defaults without having the source code. A program first calls defopen with the pathname of a file containing the default entries. Defopen returns a 0 if it is successful in opening the file, it returns the fopen failure code (errno) if the open fails. The program then calls defread with a character string, pattern. Defread reads the previously opened file from the beginning until it encounters a line beginning with pattern. Defread then returns a pointer to the first character in the line after the initial pattern characters. When all items of interest have been extracted from the opened file the program may call defopen with the name of another file to be searched, or it may call defopen with **NULL**, which closes the default file without opening another.

FILES

The XENIX convention is for a system program xyz to store its defaults (if any) in file **/etc/default/xyz**

DIAGNOSTICS

Defopen returns non-zero if the open fails. The return value is the **errno** value set by fopen(3S). Defread returns **NULL** if a default file is not open, if the indicated pattern could not be found, or if it encounters any line in the file greater than the maximum length of 128 characters.

NAME

ecvt, fcvt, gcvt - output conversion

SYNTAX

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts the value to a null-terminated string of ndigit ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by sign is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to ecvt, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by ndigits.

Gcvt converts the value to a null-terminated ASCII string in buf and returns a pointer to buf. It attempts to produce ndigit significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3)

NOTES

The return values point to static data whose content is overwritten by each call.

NAME

end, etext, edata - last locations in program

SYNTAX

```
extern end;
extern etext;
extern edata;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of etext is the first address above the program text, edata above the initialized data region, and end above the uninitialized data region.

For non-fixed stack machines, when execution begins, the program break coincides with end, but many functions reset the program break, among them the routines of brk(2), malloc(3), standard input/output (stdio(3)), the profile (-p) option of cc(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see brk(2).

SEE ALSO

brk(2), malloc(3)

NAME

exp, log, log10, pow, sqrt - exponential, logarithm, power, square root

SYNTAX

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

DESCRIPTION

Exp returns the exponential function of x.

Log returns the natural logarithm of x; log10 returns the base 10 logarithm.

Pow returns x to the y power.

Sqrt returns the square root of x.

SEE ALSO

hypot(3), sinh(3), intro(2)

DIAGNOSTICS

Exp and pow return a huge value when the correct value would overflow; errno is set to ERANGE. Pow returns 0 and sets errno to EDOM when the second argument is negative and non-integral and when both arguments are 0.

Log returns 0 when x is zero or negative; errno is set to EDOM.

Sqrt returns 0 when x is negative; errno is set to EDOM.

NAME

fclose, fflush - close or flush a stream

SYNTAX

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffers for the named stream to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

Fclose is performed automatically upon calling exit(2).

Fflush causes any buffered data for the named output stream to be written to that file. The stream remains open.

SEE ALSO

close(2), fopen(3), setbuf(3)

DIAGNOSTICS

These routines return **EOF** if stream is not associated with an output file, or if buffered data cannot be transferred to that file.

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNTAX

```
#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;
```

DESCRIPTION

Feof returns non-zero if end of file is read on the named input stream, otherwise zero.

Ferror returns non-zero when an error has occurred reading or writing the named stream, otherwise zero. Unless cleared by clearerr, the error indication lasts until the stream is closed.

Clrerr resets the error indication on the named stream.

Fileno returns the integer file descriptor associated with the stream, see open(2).

These functions are implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3), open(2)

NAME

fabs, floor, ceil - absolute value, floor, ceiling functions

SYNTAX

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double(x);
```

DESCRIPTION

Fabs returns the absolute value |x|.

Floor returns the largest integer not greater than x.

Ceil returns the smallest integer not less than x.

SEE ALSO

abs(3)

NAME

fopen, freopen, fdopen - open a stream

SYNTAX

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
char *type;
```

DESCRIPTION

Fopen opens the file named by filename and associates a stream with it. Fopen returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

"r" open for reading

"w" create for writing

"a" append: open for writing at end of file, or create for writing

Freopen substitutes the named file in place of the open stream. It returns the original value of stream. The original stream is closed.

Freopen is typically used to attach the preopened constant names, **stdin, stdout, stderr,** to specified files.

Fdopen associates a stream with a file descriptor obtained from open, dup, creat, or pipe(2). The type of the stream must agree with the mode of the open file.

SEE ALSO

open(2), fclose(3)

DIAGNOSTICS

Fopen and freopen return the pointer **NULL** if filename cannot be accessed.

NOTES

Fdopen is not portable to systems other than XENIX. Fdopen does not work properly with file descriptors 0, 1, and 2

(stdin,stdout, and **stderr,** respectively). Use freopen instead.

NAME

fread, fwrite - buffered binary input/output

SYNTAX

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

DESCRIPTION

Fread reads, into a block beginning at ptr, nitems of data of the type of *ptr from the named input stream. It returns the number of items actually read.

Fwrite appends at most nitems of data of the type of *ptr beginning at ptr to the named output stream. It returns the number of items actually written.

SEE ALSO

read(2), write(2), fopen(3), getc(3), putc(3), gets(3), puts(3), printf(3), scanf(3)

DIAGNOSTICS

Fread and fwrite return 0 upon end of file or error.

NAME

frexp, ldexp, modf - split into mantissa and exponent

SYNTAX

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

Frexp returns the mantissa of a double value as a double quantity, x, of magnitude less than 1 and stores an integer n such that value = x*2**n indirectly through eptr.

Ldexp returns the quantity value*2**exp.

Modf returns the positive fractional part of value and stores the integer part indirectly through iptr.

NAME

fseek, ftell, rewind - reposition a stream

SYNTAX

```
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes from the beginning, the current position, or the end of the file, according as ptrname has the value 0, 1, or 2.

Fseek undoes any effects of ungetc(3).

Ftell returns the current value of the offset relative to the beginning of the file associated with the named stream. It is measured in bytes on XENIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an offset for fseek.

Rewind(stream) is equivalent to fseek(stream, 0L, 0).

SEE ALSO

lseek(2), fopen(3)

DIAGNOSTICS

Fseek returns -1 for improper seeks.

NAME

getc, getchar, fgetc, getw - get character or word from stream

SYNTAX

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character from the named input stream.

Getchar() is identical to getc(stdin).

Fgetc behaves like getc, but is a genuine function, not a macro; it therefore, may be used as an argument. Also, fgetc runs more slowly than getc since a function is called, but fgetc has the advantage of taking less space per invocation, since it need not be repeatedly expanded in the text.

Getw returns the next word from the named input stream. It returns the constant **EOF** upon end of file or error, but since that is a good integer value, feof and ferror(3) should be used to check the success of getw. Getw assumes no special alignment in the file.

SEE ALSO

fopen(3), putc(3), gets(3), scanf(3), fread(3), ungetc(3)

DIAGNOSTICS

These functions return the integer constant **EOF** at end of file or upon read error.

A stop with message, 'Reading bad file', means an attempt has been made to read from a stream that has not been opened for reading by fopen.

NOTES

The end-of-file return from getchar is incompatible with that in UNIX editions 1-6.

Because it is implemented as a macro, getc treats a stream argument with side effects incorrectly. In particular,

'getc(*f++);' doesn't work sensibly.

NAME

getenv - value for environment name

SYNTAX

```
char *getenv(name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see environ(5)) for a string of the form name=value and returns value if such a string is present, otherwise 0 (NULL).

SEE ALSO

environ(5), exec(2)

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent - get group file entry

SYNTAX

```
#include <grp.h>

struct group *getgrent();

struct group *getgrgid(gid) int gid;

struct group *getgrnam(name) char *name;

int setgrent();

int endgrent();
```

DESCRIPTION

Getgrent, getgrgid and getgrnam each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

```
struct  group { /* see getgrent(3) */
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};
```

The members of this structure are:

gr_name
: The name of the group.

gr_passwd
: The encrypted password of the group.

gr_gid
: The numerical group-ID.

gr_mem
: Null-terminated vector of pointers to the individual member names.

Getgrent simply reads the next line while getgrgid and getgrnam search until a matching gid or name is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

A call to setgrent has the effect of rewinding the group file to allow repeated searches. Endgrent may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3), getpwent(3), group(5)

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getlogin - get login name

SYNTAX

char *getlogin();

DESCRIPTION

Getlogin returns a pointer to the login name as found in /etc/utmp. It may be used in conjunction with getpwnam to locate the correct password file entry when the same userid is shared by several login names.

If getlogin is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to first call getlogin and if it fails, to call getpwuid.

FILES

/etc/utmp

SEE ALSO

getpwent(3), getgrent(3), utmp(5)

DIAGNOSTICS

Returns NULL (0) if name not found.

NOTES

The return values point to static data whose content is overwritten by each call.

NAME

getpass - read a password

SYNTAX

```
char *getpass(prompt)
char *prompt;
```

DESCRIPTION

Getpass reads a password from the file /dev/tty, or if that cannot be opened, from the standard input, after prompting with the null-terminated string prompt and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

/dev/tty

SEE ALSO

crypt(3)

NOTES

The return value points to static data whose content is overwritten by each call.

NAME

getpw - get name from UID

SYNTAX

```
getpw(uid, buf)
char *buf;
```

DESCRIPTION

Getpw searches the password file for the (numerical) uid, and fills in buf with the corresponding line; it returns non-zero if uid could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error, otherwise 0.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent - get password file entry

SYNTAX

```
#include <pwd.h>

struct passwd *getpwent();

struct passwd *getpwuid(uid) int uid;

struct passwd *getpwnam(name) char *name;

int setpwent();

int endpwent();
```

DESCRIPTION

Getpwent, getpwuid and getpwnam each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

```
struct  passwd { /* see getpwent(3) */
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        int     pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

The fields pw_quota and pw_comment are unused; the others have meanings described in passwd(5).

Getpwent reads the next line (opening the file if necessary); setpwent rewinds the file; endpwent closes it.

Getpwuid and getpwnam search from the beginning until a matching uid or name is found (or until EOF is encountered).

FILES

/etc/passwd

SEE ALSO

getlogin(3), getgrent(3), passwd(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

NAME

gets, fgets - get a string from a stream

SYNTAX

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Gets reads a string into s from the standard input stream stdin. The string is terminated by a newline character, which is replaced in s by a null character. Gets returns its argument.

Fgets reads n-1 characters, or up to a newline character, whichever comes first, from the stream into the string s. The last character read into s is followed by a null character. Fgets returns its first argument.

SEE ALSO

puts(3), getc(3), scanf(3), fread(3), ferror(3)

DIAGNOSTICS

Gets and fgets return the constant pointer **NULL** upon end of file or error.

NOTES

Gets deletes a newline, fgets keeps it, all in the name of backward compatibility.

NAME

hypot, cabs - euclidean distance

SYNTAX

```
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;
```

DESCRIPTION

Hypot and cabs return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

exp(3) for sqrt

NAME

j0, jl, jn, y0, yl, yn - bessel functions

SYNTAX

```
#include <math.h>

double j0(x)
double x;

double jl(x)
double x;

double jn(n, x);
double x;

double y0(x)
double x;

double yl(x)
double x;

double yn(n, x)
double x;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause y0, yl, and yn to return a huge negative value and set errno to EDOM.

NAME

l3tol, ltol3 - convert between 3-byte integers and long integers

SYNTAX

```
l3tol(lp, cp, n)
long *lp;
char *cp;

ltol3(cp, lp, n)
char *cp;
long *lp;
```

DESCRIPTION

L3tol converts a list of n three-byte integers packed into a character string pointed to by cp into a list of long integers pointed to by lp.

Ltol3 performs the reverse conversion from long integers (lp) to three-byte integers (cp).

These functions are useful for file-system maintenance; disk addresses are three bytes long.

SEE ALSO

filsys(5)

NAME

malloc, free, realloc, calloc - main memory allocator

SYNTAX

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and free provide a simple general-purpose memory allocation package. Malloc returns a pointer to a block of at least size bytes beginning on a word boundary.

The argument to free is a pointer to a block previously allocated by malloc; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by malloc is overrun or if some random number is handed to free.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls sbrk (see break(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Realloc also works if ptr points to a block freed since the last call of malloc, realloc or calloc; thus sequences of free, malloc and realloc can exploit the search strategy of malloc to do storage compaction.

Calloc allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for

storage of any type of object.

DIAGNOSTICS

Malloc, realloc and calloc return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. Malloc may be recompiled to check the arena very stringently on every transaction; see the source code.

NOTES

When realloc returns 0, the block pointed to by ptr may be destroyed.

NAME

mktemp - make a unique file name

SYNTAX

```
char *mktemp(template)
char *template;
```

DESCRIPTION

Mktemp replaces template with a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with a letter and the current process id.

SEE ALSO

getpid(2)

NOTES

It is possible to run out of letters.

NAME

monitor - prepare execution profile

SYNTAX

```
monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short buffer[ ];
```

DESCRIPTION

Monitor is an interface to profil(2). Lowpc and highpc are the addresses of two functions; buffer is the address of a (user supplied) array of bufsize short integers. Monitor arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of lowpc and the highest is just below highpc. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext();
...
monitor((int)2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text, see end(3).

To stop execution monitoring and write the results on the file mon.out, use

```
monitor(0);
```

then prof(1) can be used to examine the results.

FILES

mon.out

SEE ALSO

prof(1), profil(2)

NOTES

The prof(1) program may require the buffer size to be equal to or smaller than the program size.

NAME

itom, madd, msub, mult, mdiv, min, mout, pow, gcd, rpow - multiple precision integer arithmetic

SYNTAX

```
typedef struct { int len; short *val; } mint;

madd(a, b, c)
msub(a, b, c)
mult(a, b, c)
mdiv(a, b, q, r)
min(a)
mout(a)
pow(a, b, m, c)
gcd(a, b, c)
rpow(a, b, c)
msqrt(a, b, r)
mint *a, *b, *c, *m, *q, *r;

sdiv(a, n, q, r)
mint *a, *q;
short *r;

mint *itom(n)
```

DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type mint. Pointers to a mint should be initialized using the function itom, which sets the initial value to n. After that space is managed automatically by the routines.

madd, msub, mult, assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. mdiv assigns the quotient and remainder, respectively, to its third and fourth arguments. sdiv is like mdiv except that the divisor is an ordinary integer. msqrt produces the square root and remainder of its first argument. rpow calculates a raised to the power b, while pow calculates this reduced modulo m. min andmout do decimal input and output.

The functions ane obtained with the loader option -lmp.

DIAGNOSTICS

Illegal operations and running out of memory produce messages and core images.

NAME

nlist - get entries from name list

SYNTAX

```
#include <a.out.h>
nlist(filename, nl)
char *filename;
struct nlist nl[ ];
```

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. "nl" consists of an array of "nlist" structures containing names, types, and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted into the next two fields. If the name is not found, both entries are set to zero. Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list structure nl consists of an array of nlist structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted into the next two fields. If the name is not found, both entries are set to zero. See a.out(5) for a discussion of the nlist structure.

The two understood formats are 'x.out' and 'a.out'.

If two or more symbols happen to match the name given to nlist, then the type and value used will be those of the last symbol found.

SEE ALSO

a.out(5)

DIAGNOSTICS

Nlist returns -1 and sets all type entries to zero if the file cannot be read, is not an object file, or contains an invalid name list. If the value in the namelist cannot fit in an integer field, then the type and value are set to zero and -1 is returned after completion of the name list search. Otherwise, nlist returns zero. A return value of zero does not indicate whether or not any or all of the given symbols were found. Otherwise, nlist returns zero.

NAME

perror, sys_errlist, sys_nerr - system error messages

SYNTAX

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

DESCRIPTION

Perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string s is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable errno (see intro(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings sys errlist is provided; errno can be used as an index in this table to get the message string without the newline. Sys nerr is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2)

NAME

plot: openpl et al. - graphics interface

SYNTAX

```
openpl( )

erase( )

label(s) char s[ ];

line(xl, yl, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, xl, yl)

move(x, y)

cont(x, y)

point(x, y)

linemod(s) char s[ ];

space(x0, y0, xl, yl)

closepl( )
```

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See plot(5) for a description of their effect. Openpl must be used before any of the others to open the device for writing. Closepl flushes the output.

String arguments to label and linemod are null-terminated, and do not contain newlines.

Various flavors of these functions exist for different output devices. They are obtained by the following ld(1) options:

-lplot	device-independent graphics stream on standard output for plot(1) filters
-l300	GSI 300 terminal
-l300s	GSI 300S terminal
-l450	DASI 450 terminal
-l4014	Tektronix 4014 terminal

SEE ALSO

plot(5), plot(1), graph(1)

NAME

popen, pclose - initiate I/O to/from a process

SYNTAX

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;
```

DESCRIPTION

The arguments to popen are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by popen should be closed by pclose, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

SEE ALSO

pipe(2), fopen(3), fclose(3), system(3), wait(2)

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or the Shell cannot be accessed.

Pclose returns -1 if stream is not associated with a 'popened' command.

NOTES

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with fflush, see fclose(3).

NAME

printf, fprintf, sprintf - formatted output conversion

SYNTAX

```
#include <stdio.h>

printf(format [, arg ] ... )
char *format;

fprintf(stream, format [, arg ] ... )
FILE *stream;
char *format;

sprintf(s, format [, arg ] ... )
char *s, *format;
```

DESCRIPTION

Printf places output on the standard output stream stdout. Fprintf places output on the named output stream. Sprintf places 'output' in the string s, followed by the character '\0'.

Each of these functions converts, formats, and prints its arguments after the format string under control of the format string argument. The format argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive arg printf.

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign '-' which specifies left adjustment of the converted value in the indicated field;
- an optional digit string specifying a field width; if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- an optional period '.' which serves to separate the field width from the next digit string;
- an optional digit string specifying a precision which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;

- the character **l** specifying that a following d, o, x, or **u** corresponds to a long integer arg. (A capitalized conversion code accomplishes the same thing.)

- a character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer arg supplies the field width or precision.

The conversion characters and their meanings are

dox The integer arg is converted to decimal, octal, or hexadecimal notation respectively.

f The float or double arg is converted to decimal notation in the style '[-]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e The float or double arg is converted in the style '[-]d.ddde±dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double arg is printed in style d, in style **f**, or in style **e**, whichever gives full precision in minimum space.

c The character arg is printed. Null characters are ignored.

s Arg is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer arg is converted to decimal and printed (the result will be in the range 0 to 65535).

% Print a '%'; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by printf are printed by putc(3).

Examples

To print a date and time in the form 'Sunday, July 3, 10:02', where weekday and month are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day,
    hour, min);
```

To print pi to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

putc(3), scanf(3), ecvt(3)

NOTES

Very wide fields (>128 characters) fail.

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNTAX

```
#include <stdio.h>

putc(c, stream)
char c;
FILE *stream;

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;
```

DESCRIPTION

Putc appends the character c to the named output stream. It returns the character written.

Putchar(c) is defined as putc(c, stdout).

Fputc behaves like putc, but is a genuine function rather than a macro. It may be used to save on object text.

Putw appends word (i.e. **int**) w to the output stream. It returns the word written. Putw neither assumes nor causes special alignment in the file.

The standard stream stdout is normally buffered if and only if the output does not refer to a terminal; this default may be changed by setbuf(3). The standard stream stderr is by default unbuffered unconditionally, but use of freopen (see fopen(3)) will cause it to become buffered; setbuf, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. Fflush (see fclose(3)) may be used to force the block out early.

SEE ALSO

fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, ferror(3) should be used to detect putw errors.

NOTES

Because it is implemented as a macro, putc treats a stream argument with side effects improperly. In particular 'putc(c, *f++);' doesn't work sensibly.

NAME

puts, fputs - put a string on a stream

SYNTAX

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts copies the null-terminated string s to the standard output stream stdout and appends a newline character.

Fputs copies the null-terminated string s to the named output stream.

Neither routine copies the terminal null character.

SEE ALSO

fopen(3), gets(3), putc(3), printf(3), ferror(3)
fread(3) for fwrite

NOTES

Puts appends a newline, fputs does not, all in the name of backward compatibility.

NAME

qsort - quicker sort

SYNTAX

```
qsort(base, nel, width, compar)
char *base;
int (*compar)( );
```

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1)

NAME

rand, srand - random number generator

SYNTAX

```
srand(seed)
int seed;

rand( )
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator with period 2^32 to return successive pseudo-random numbers in the range from 0 to (2^15)-1.

The generator is reinitialized by calling srand with 1 as argument. It can be set to a random starting point by calling srand with whatever you like as argument.

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNTAX

```
#include <stdio.h>

scanf(format [ , pointer ] . . . )
char *format;

fscanf(stream, format [ , pointer ] . . . )
FILE *stream;
char *format;

sscanf(s, format [ , pointer ] . . . )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream stdin. Fscanf reads from the named input stream. Sscanf reads from the character string s. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string format, described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.

2. An ordinary character (not %) which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

% a single '%' is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

o an octal integer is expected; the corresponding argument should be a integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try '%1s'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

e,f a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, **o** and **x** may be capitalized or preceeded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e** or **f** may be capitalized or preceded by l to indicate a pointer to **double** rather than to **float.** The

conversion characters **d**, **o** and **x** may be preceeded by **h** to indicate a pointer to **short** rather than to **int**.

The scanf functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf( "%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E-1  thompson
```

will assign to i the value 25, x the value 5.432, and name will contain 'thompson\0'. Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to i, 789.0 to x, skip '0123', and place the string '56\0' in name. The next call to getchar will return 'a'.

SEE ALSO

atof(3), getc(3), printf(3)

DIAGNOSTICS

The scanf functions return **EOF** on end of input, and a short count for missing or illegal data items.

NOTES

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf - assign buffering to a stream

SYNTAX

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array buf to be used instead of an automatically allocated buffer. If buf is the constant pointer **NULL**, input/output will be completely unbuffered.

A manifest constant **BUFSIZ** tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from malloc(3) upon the first getc or putc(3) on the file, except that output streams directed to terminals, and the standard error stream stderr are normally not buffered.

SEE ALSO

fopen(3), getc(3), putc(3), malloc(3)

NAME

setjmp, longjmp - non-local goto

SYNTAX

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in env for later use by longjmp. It returns value 0.

Longjmp restores the environment saved by the last call of setjmp. It then returns in such a way that execution continues as if the call of setjmp had just returned the value val to the function that invoked setjmp, which must not itself have returned in the interim. All accessible data have values as of the time longjmp was called except for register variables whose values are undefined.

SEE ALSO

signal(2)

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNTAX

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

DESCRIPTION

Sin, cos and tan return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sin in the range -J/2 to J/2.

Acos returns the arc cosine in the range 0 to J.

Atan returns the arc tangent of x in the range -J/2 to J/2.

Atan2 returns the arc tangent of x/y in the range -J to J.

DIAGNOSTICS

Arguments of magnitude greater than 1 cause asin and acos to return value 0; errno is set to EDOM. The value of tan at its singular points is a huge number, and errno is set to ERANGE.

NOTES

The value of tan for arguments greater than about 2**31 is garbage.

NAME

sinh, cosh, tanh - hyperbolic functions

SYNTAX

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

Sinh and cosh return a huge value of appropriate sign when the correct value would overflow.

NAME

sleep - suspend execution for interval

SYNTAX

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an alarm clock signal and pausing until it (or some other signal) occurs. The previous state of this signal is saved and restored. The calling program may have set up an alarm signal before calling sleep. If the sleep time exceeds the time until the alarm signal occurs, then the process sleeps only until the signal would have occurred, and the caller's alarm catch routine is executed just before the sleep routine returns. However, if the sleep time is less than the time until such alarm, the prior alarm time is reset to go off at the same time it would have without intervening sleep.

SEE ALSO

alarm(2), pause(2), signal(2)

NAME

stdio - standard buffered input/output package

SYNTAX

```
#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION

The functions described in Sections 3S constitute an efficient user-level buffering scheme. The in-line macros getc and putc(3) handle characters quickly. The higher level routines gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf, fwrite all use getc and putc; they can be freely intermixed.

A file with associated buffering is called a stream, and is declared to be a pointer to a defined type **FILE.** Fopen(3) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant 'pointer' **NULL** (0) designates no stream at all.

An integer constant **EOF** (-1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file <stdio.h> of pertinent macro definitions. The functions and constants mentioned in sections labeled 3S are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: getc, getchar, putc, putchar, feof, ferror, fileno.

SEE ALSO

open(2), close(2), read(2), write(2)

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **FILE** pointer has not been initialized with fopen, input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or unintelligible **FILE** data.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - string operations

SYNTAX

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string s2 to the end of string s1. Strncat copies at most n characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as s1 is lexicographically greater than, equal to, or less than s2. Strncmp makes the same comparison but looks at at most n characters.

Strcpy copies string s2 to s1, stopping after the null character has been moved. Strncpy copies exactly n characters, truncating or null-padding s2; the target may not be null-terminated if the length of s2 is n or more. Both return s1.

Strlen returns the number of non-null characters in s.

Index (rindex) returns a pointer to the first (last) occurrence of character c in string s, or zero if c does not occur in the string.

NOTES

Strcmp uses native character comparison, which may be either signed or unsigned depending on the machine.

NAME

swab - swap bytes

SYNTAX

```
swab(from, to, nbytes)
char *from, *to;
```

DESCRIPTION

Swab copies nbytes bytes pointed to by from to the position pointed to by to, exchanging adjacent even and odd bytes. It is useful for carrying binary data between different machines. Nbytes should be even.

NAME

system - issue a shell command

SYNTAX

```
system(string)
char *string;
```

DESCRIPTION

System causes the string to be given to sh(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

popen(3), exec(2), wait(2)

DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operation routines

SYNTAX

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base termcap(5). These are low level routines; see curses(3) for a higher level package.

Tgetent extracts the entry for terminal name into the buffer at bp. Bp should be a character buffer of size 1024 and must be retained through all subsequent calls to tgetnum, tgetflag, and tgetstr. Tgetent returns -1 if it cannot open the termcap file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a pathname rather than /etc/termcap. This can speed up entry into programs that call tgetent, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file /etc/termcap.

Tgetnum gets the numeric value of capability id, returning -1 if is not given for the terminal. Tgetflag returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. Tgetstr gets the string value of capability id, placing it in the buffer at area, advancing the area pointer. It decodes the abbreviations for this field described in termcap(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from cm to go to column destcol in line destline. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since tgoto may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a **%** sequence is given which is not understood, then tgoto returns OOPS.

Tputs decodes the leading padding information of the string cp; affcnt gives the number of lines affected by the operation, or 1 if this is not applicable, outc is a routine which is called with each character in turn. The external variable ospeed should contain the output speed of the terminal as encoded by stty (2). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

FILES

/usr/lib/libtermcap.a	-ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NAME

ttyname, isatty, ttyslot - find name of a terminal

SYNTAX

char *ttyname(fildes)

isatty(fildes)

ttyslot()

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor fildes.

Isatty returns 1 if fildes is associated with a terminal device, 0 otherwise.

Ttyslot returns the number of the entry in the ttys(5) file for the control terminal of the current process.

FILES

/dev/*
/etc/ttys

SEE ALSO

ioctl(2), ttys(5)

DIAGNOSTICS

Ttyname returns a null pointer (0) if fildes does not describe a terminal device in directory '/dev'.

Ttyslot returns 0 if '/etc/ttys' is inaccessible or if it cannot determine the control terminal.

NOTES

The return value points to static data whose content is overwritten by each call.

NAME

ungetc - push character back into input stream

SYNTAX

```
#include <stdio.h>

int ungetc(c, stream)
char c
FILE *stream;
```

DESCRIPTION

Ungetc pushes the character c back on an input stream. That character will be returned by the next getc call on that stream. Ungetc returns c.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push **EOF** are rejected.

Fseek(3) erases all memory of pushed back characters.

SEE ALSO

getc(3), setbuf(3), fseek(3)

DIAGNOSTICS

Ungetc returns **EOF** if it can't push a character back.

NAME

intro - introduction to special files

DESCRIPTION

This section describes various special files that refer to specific peripherals and XENIX device drivers. The names of the entries are generally derived from the manufacturer's names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding XENIX device driver are discussed where applicable.

NOTES

While the names of the entries generally refer to hardware names, in certain cases these names are seemingly arbitrary for various historical reasons.

NAME

console - serial terminal interface

DESCRIPTION

The standard Model 16 supports three terminals, one of which, the system console, does not use a true serial electrical interface. Because the console is not truly serial, it ignores requests to change line speeds. Except for the console, the serial lines behave as documented in tty(4).

FILES

/dev/console, /dev/tty01, /dev/tty02

SEE ALSO

tty(4)

NOTES

The keys of the console terminal are specially mapped; some control-key combinations produce unexpected codes. This key mapping is documented elsewhere. Full support for modem control signals is not implemented.

NAME

fd - floppy disk

DESCRIPTION

The floppy disk interface can deal with a variety of disk formats; multiple densities and track and sector sizes are supported. When a floppy drive becomes active, the disk format is automatically sensed; the user can use one device name to deal with any supported disk format.

The octal representation of the minor device number is encoded 0dp, where d is a physical drive number from zero to three, inclusive, and p is a pseudodrive (subsection) within a physical unit. (Physical drive numbers four to seven represent hard disks.) The descriptions of the pseudodisks are:

disk	description
0	entire disk, except boot track (if any)
1	file system on disk, if any, else invalid
2	swap area on disk, if any, else invalid
3	boot track, if any, else invalid
4	reserved
5	reserved
6	reserved
7	reserved

The swap area on a disk is defined to begin at the end of the file system (if any) on the disk and include the rest of the disk. The names of the boot-track partitions conventionally have the letters 'bt' inserted before the drive number.

Standard Model-16 XENIX double-sided floppies hold 2448 blocks, not counting the boot track. Standard single-sided floppies hold 1216 blocks. The boot track on standard floppies holds 26 sectors, each of 128 bytes.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O, the buffer must begin on a word boundary, the disk address must be an exact multiple of 512, and the number of bytes in the I/O request must be a multiple of 512.

If a disk drive holds the current root filesystem, then 'root' and 'rroot' are conventionally used to refer to the normal and raw interfaces to the filesystem pseudodisk of the given disk drive. Similarly, if a disk drive is being used as the current swap device, then 'swap' is conventionally used to refer to the swap pseudodisk of that drive.

FILES

/dev/fd?, /dev/rfd?, /dev/fdbt?, /dev/rfdbt?, /dev/swap, /dev/root, /dev/rroot

SEE ALSO

hd(4), physio(5)

NOTES

In programs that are likely to access raw devices, read, write and lseek(2) should always deal in 512-byte multiples.

NAME

hd - hard disk

DESCRIPTION

The hard disk interface can deal with more than one geometry hard disk. Each hard disk must have been properly formatted and initialized with the **diskutil** standalone utility for the hard disk interface to function properly.

The octal representation of the minor device number is encoded 0dp, where d is a number from four to seven, inclusive, corresponding to physical hard disk drives zero to three, and p is a pseudodrive (subsection) within a physical unit. (Values of d from zero to three represent floppy disks.) The descriptions of the pseudodisks are:

disk	description
0	entire disk, except boot and diagnostic tracks
1	file system on disk, if any, else invalid
2	swap area on disk, if any, else invalid
3	boot and diagnostic tracks
4	reserved
5	reserved
6	reserved
7	reserved

The swap area on a disk is defined to begin at the end of the file system (if any) on the disk and include the rest of the disk. The names of the boot-track partitions conventionally have the letters 'bt' inserted before the drive number. The boot-track partitions include two tracks; the first of these tracks is the actual boot track, and the second is reserved for use by diagnostic software.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records.

A 'raw' interface provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.' In raw I/O the buffer must begin on a word boundary, the disk address must be an exact multiple of 512, and the number of bytes in the I/O request must be a multiple of 512.

If a disk drive holds the current root filesystem, then 'root' and 'rroot' are conventionally used to refer to the normal and raw interfaces to the filesystem pseudodisk of the given disk drive. Similarly, if a disk drive is being

used as the current swap device, then 'swap' is conventionally used to refer to the swap pseudodisk of that drive.

FILES

/dev/hd?, /dev/rhd?, /dev/hdbt?, /dev/rhdbt?, /dev/swap, /dev/root, /dev/rroot

SEE ALSO

fd(4), physio(5)

NOTES

In programs that are likely to access raw devices, read, write and lseek(2) should always deal in 512-byte multiples.

NAME

lp - line printer

DESCRIPTION

Lp and rlp provide the interface to a standard Radio Shack parallel printer attached to the parallel output port on the Model 16. Rlp is the 'raw' line printer device, which passes eight bit characters directly to the printer without processing. The 'raw' printer can be treated by software as if it were a write-only serial terminal, initialized to be in raw mode when opened. The minor device number of the 'raw' printer device is 3.

Lp is the 'logical' printer device. Characters sent to the 'logical' printer are specially processed. Line-feed and carriage-return characters are treated identically, and cause a line-feed character to be sent to the printer if the current print column is equal to zero; if the current column number is non-zero, a carriage-return is sent.

Tab characters are expanded into an appropriate number of spaces, and form-feeds are expanded into the appropriate number of carriage-returns and line-feeds. The driver assumes that tab stops occur at eight character intervals; its default number of lines per page is 66.

Whenever the 'logical' line printer is opened, the system's idea of the current print line and column are reset to zero; when the 'logical' line printer is closed, the driver forces the printer to its idea of start of page. The driver's idea of current print line and column is also reset whenever a program changes the number of lines per page.

Only the super-user can open a line printer device while the line printer is already open. Also, only the super-user can change the system's idea of number of lines per page.

FILES

/dev/lp, /dev/rlp

SEE ALSO

lpr(1), tty(4)

NOTES

The 'logical' line printer's idea of current print line and column may be disrupted by various escape sequences and control characters. In particular, the 'logical' device assumes that the backspace character causes the print-head to back up one column if it is not in column zero; if a specific printer does not treat the backspace character in this fashion, then a backspace will cause the printer driver's idea of the current print column to become

LP(4)

inaccurate.

NAME

null - data sink

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

TTY(4)

NAME

tty - general terminal interface

DESCRIPTION

This section describes both a particular special file, and the general nature of the terminal interface.

The file /dev/tty is, in each process, a synonym for the control terminal associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

As for terminals in general: all of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of the interface.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by init and become a user's input and output file. The very first terminal file open in a process becomes the control terminal for that process. The control terminal plays a special role in handling quit or interrupt signals, as discussed below. The control terminal is inherited by a child process during a fork, even if the control terminal is closed. The set of processes that thus share a control terminal is called a process group; all members of a process group receive certain signals together, see INTERRUPT below and kill(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information. There are special modes, discussed below, that permit the program to read each character as typed without

waiting for a full line.

During input, erase and kill processing is normally done. By default, the character '^H' erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. By default, the character '^U' kills the entire line up to the point where it was typed, but not beyond an EOT. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that may have been done. Either '^H' or '^U' may be entered literally by preceding it by '\'; the erase or kill character remains, but the '\' disappears. These two characters may be changed to others.

When desired, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. In addition, the following escape sequences can be generated on output and accepted on input:

for	use
`	\'
|	\!
~	\^
{	\(
}	\)

Certain ASCII control characters have special meaning. These characters are not passed to a reading program except in raw mode where they lose their special character. Also, it is possible to change these characters from the default; see below.

EOT (Control-D) may be used to generate an end of file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication.

INTERRUPT
(usually Control-C, Break, DEL, or Rubout) This is not passed to a program but generates an interrupt signal which is sent to all processes with the associated control terminal. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See signal(2).

FS (Control-\ or control-shift-L) generates the quit

signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated.

DC3 (Control-S) delays all printing on the terminal until something is typed in.

DC1 (Control-Q) restarts printing after DC3 without generating any input to a program.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a hangup signal is sent to all processes with the terminal as control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file on their input can terminate appropriately when hung up on.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is always generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals that respond to it from hanging up.

Several ioctl(2) calls apply to terminals. Most of them use the following structure, defined in <sgtty.h>:

```
struct sgttyb {
     char sg_ispeed;
     char sg_ospeed;
     char sg_erase;
     char sg_kill;
     int  sg_flags;
};
```

The sg_ispeed and sg_ospeed fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <sgtty.h>.

```
B0      0    (hang up dataphone)
B50     1    50 baud
```

B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The sg erase and sg kill fields of the argument structure specify the erase and kill characters respectively. (Defaults are Control-H and Control-U.)

The sg flags field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	

TTY(4)

EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is unimplemented and is 0.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file indicator (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially. There are no delays and no echoing, and no replacement of one character for another; characters are a full 8 bits for both input and output (parity is up to the program).

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a new-line function).

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line, but quit and interrupt work, and output delays, case-translation, CRMOD, XTABS, ECHO, and parity work normally. On the other hand there is no erase or kill, and no special treatment of \ or EOT.

TANDEM mode causes the system to produce a stop character (default DC3) whenever the input queue is in danger of overflowing, and a start character (default DC1) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is actually another machine that obeys the conventions.

Several ioctl calls have the form:

```
#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;
```

The applicable codes are:

TIOCGETP
: Fetch the parameters associated with the terminal, and store in the pointed-to structure.

TIOCSETP
: Set the parameters according to the pointed-to structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN
: Set the parameters but do not delay or flush input. Switching out of RAW or CBREAK mode may cause some garbage input.

With the following codes the arg is ignored.

TIOCEXCL
: Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL
: Turn off "exclusive-use" mode.

TIOCHPCL
: When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH
All characters waiting in input or output queues are flushed.

The following codes affect characters that are special to the terminal interface. The third argument is a pointer to the following structure, defined in <sgtty.h>:

```
struct tchars {
    char    t_intrc;        /* interrupt */
    char    t_quitc;        /* quit */
    char    t_startc;       /* start output */
    char    t_stopc;        /* stop output */
    char    t_eofc;         /* end-of-file */
    char    t_brkc;         /* input delimiter (like nl) */
};
```

The default values for these characters are DEL, FS, DC1, DC3, EOT, and -1. A character value of -1 eliminates the effect of that character. The t_brkc character, by default -1, acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (especially erase and kill) identical.

The calling codes for the tchars structure are:

TIOCSETC
Copy the pointed to tchars structure into the systems working copy, so that the values in the structure take effect, replacing (and destroying) the previous values.

TIOCGETC
Get the special character structure and store it in the pointed to tchars structure.

FILES
/dev/tty
/dev/tty*
/dev/console

SEE ALSO
getty(8), stty (1), signal(2), ioctl(2)

NOTES
Half-duplex terminals are not supported.

The terminal handler has clearly entered the race for ever-greater complexity and generality.

NAME

intro - introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys.**

NAME

a.out - assembler and link editor output

SYNOPSIS

```
#include <a.out.h>
#include <sys/relsym.h>
#include <sys/relsym86.h>
```

DESCRIPTION

A.out is the output file of the assembler as(1) and the link editor ld(1). Both programs make a.out executable if there were no errors and no unresolved external references.

A.out is the name given to all object files by default. In this context, when a file is referred to as being in x.out format, it contains an x.out header and may contain any of several symbol table and relocation record formats.

The a.out.h include file contains the header and extended header declarations and defined values for the header fields. It also contains declarations for the portable symbol structures used by library routines to access symbol tables.

The sys/relsym.h include file contains declarations and defined values for the symbol table and relocation structures used in an x.out file. The sys/relsym86.h include file contains declarations and descriptions of the relocation and symbol formats used in 8086 object files.

An x.out object file has seven sections: a header, an extended header, text, data, symbol table, text relocation records, and data relocation records (in that order). The symbols and relocation records may be empty if the program was loaded with the appropriate options to ld(1) or if they have been removed by strip(1).

When an x.out file is loaded into core for execution, three logical segments are set up: the text segment, the data segment, and stack segment. The data segment contains intialized data followed by bss, or blank storage space; bss is initialized to all 0. The text segment begins at the text relocation base (not necessarily 0) in the core image; the header itself is not loaded. If the text segment is to be impure (not write-protected), the data segment is immediately contiguous with the text segment.

If the text segment is pure, the data segment begins at the first page boundary following the text segment, and the text segment is not writeable by the program; if other processes are executing the same file, they will share the text segment. On some machines, the text and data segment locations

may be reversed.

If the text and data segments are separate (separate I & D), the text and data segments may begin at unrelated locations.

The stack segment will be located by the Xenix kernel, and on non-fixed stack machines is extended automatically as required. The data segment is only extended as requested by brk(2).

HEADER

In the header, the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

Layout information as given in the include file is:

```
struct xexec {                  /* x.out header */
        unsigned short x_magic;   /* magic number */
        unsigned short x_ext;     /* extended header size */
        long           x_text;    /* text size */
        long           x_data;    /* data size */
        long           x_bss;     /* bss size */
        long           x_syms;    /* symbol table size */
        long           x_reloc;   /* relocation size */
        long           x_entry;   /* entry point */
        char           x_cpu;     /* cpu type */
        char           x_relsym;  /* reloc & symbol format */
        unsigned short x_renv;    /* run-time environment */
};
```

Definition for the x.out magic number. The presence of this value in the first two bytes of a file indicates the file is in x.out format.

```
#define X_MAGIC    0x0206
```

Definitions for x_cpu. The first two defined bits are set if the bytes or words in the x.out header, symbol table and relocation records are not stored in the same order as on a PDP-11. The second group indicates the cpu for which the file was compiled or assembled.

```
#define XC_BSWAP  0x80    /* bytes swapped */
#define XC_WSWAP  0x40    /* words swapped */

#define XC_NONE   0x00    /* none */
#define XC_PDP11  0x01    /* PDP-11 */
#define XC_23     0x02    /* 23fixed PDP-11 */
#define XC_Z8K    0x03    /* Z8000 */
#define XC_8086   0x04    /* 8086 */
```

```
#define XC_68K     0x05     /* M68000 */
#define XC_Z80     0x06     /* Z80 */
#define XC_VAX     0x07     /* VAX 780/750 */
#define XC_CPU     0x3f     /* cpu mask */
```

Definitions for x_relsym. The first group indicates the type of relocation attached, the second indicates the type of symbol table.

```
#define XR_RXOUT     0x00     /* x.out long form */
#define XR_RXEXEC    0x10     /* x.out short form */
#define XR_RBOUT     0x20     /* b.out format */
#define XR_RAOUT     0x30     /* a.out format */
#define XR_R86REL    0x40     /* 8086 relocatable */
#define XR_R86ABS    0x50     /* 8086 absolute */
#define XR_REL       0xf0     /* relocation mask */

#define XR_SXOUT     0x00     /* struct sym */
#define XR_SBOUT     0x01     /* struct bsym */
#define XR_SAOUT     0x02     /* struct asym */
#define XR_S86REL    0x03     /* 8086 relocatable */
#define XR_S86ABS    0x04     /* 8086 absolute */
#define XR_SUCBVAX   0x05     /* string table */
#define XR_SYM       0x0f     /* symbol mask */
```

Definitions for x_renv, the run-time environment. The first group indicates the version of Xenix for which the file was compiled. The XE_LTEXT and/or XE_LDATA bits are set if text and/or data addresses require more than sixteen bits. Other bits are set to describe the configuration of the text and data segments, to indicate if the fixed stack size field in the extended header is valid, and to indicate whether it is executable or linkable.

```
#define XE_V2        0x4000     /* version 2.x */
#define XE_VERS      0xc000     /* version mask */

#define XE_RES       0x0080     /* reserved */
#define XE_LTEXT     0x0040     /* large model text */
#define XE_LDATA     0x0020     /* large model data */
#define XE_OVER      0x0010     /* text overlay */
#define XE_FS        0x0008     /* fixed stack */
#define XE_PURE      0x0004     /* pure text */
#define XE_SEP       0x0002     /* separate I & D */
#define XE_EXEC      0x0001     /* executable */
```

EXTENDED HEADER

The first two fields contain the sizes of text and data relocation attached to the file. The next two contain the base addresses for which text and data have already been relocated. The fifth field is used on fixed stack machines to indicate the size of the stack segment required for

execution.

```
struct  xext {
        long    xe_trsize;      /* text rel size */
        long    xe_drsize;      /* data rel size */
        long    xe_tbase;       /* text base */
        long    xe_dbase;       /* data base */
        long    xe_stksize;     /* stack size */
};
```

SYMBOL TABLE

The standard x.out symbol table (XR_SXOUT) is discussed here. For declarations of other supported symbol types, see the sys/relsym.h include file. Some flag values are only used internally by utility programs, and will not be found in object files.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the link editor ld(1) as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

Each symbol in the table has the structure given below, followed immediately by its name in the form of a null terminated string. The length of the symbol name, including the null terminator, must be no greater than SYMLENGTH. No effort is made to word align subsequent structures in the symbol table.

The first field encodes the symbol type, the last contains the value (usually the core address) of the symbol. The structure has been padded to allow portable use.

```
struct sym {
        unsigned short  s_type;
        unsigned short  s_pad;
        long            s_value;
};

#define SYMLENGTH    50
```

Definitions for s_type:

```
#define S_UNDEF   0x0000    /* undefined */
#define S_ABS     0x0001    /* absolute */
#define S_TEXT    0x0002    /* text */
#define S_DATA    0x0003    /* data */
#define S_BSS     0x0004    /* bss */
#define S_COMM    0x0005    /* internal */
#define S_REG     0x0006    /* register */
#define S_COMB    0x0007    /* internal */
#define S_TYPE    0x001f    /* mask */

#define S_FN      0x001f    /* file name */
#define S_EXTERN  0x0020    /* external bit */
```

The nlist structure, provided for compatibility with nlist(3).

```
struct  nlist {
        char      n_name[8];  /* symbol name */
        int       n_type;     /* type flag */
        unsigned  n_value;    /* value */
};
```

RELOCATION

If relocation information is present, it takes one of two forms. For linkable object files, the long form (XR_RXOUT) is used to enable references between modules to be resolved when linked together.

The first field encodes the segment referenced, the size (number of bytes) to relocate, and whether the relocation is relative. The second contains the symbol id, an index into the symbol table. The first symbol in the symbol table is referenced with 0. The symbol id is used to obtain the symbol value in order to perform external relocation. The last field contains the position within the current segment at which relocation is to be performed.

```
struct reloc {
        unsigned short  r_desc;     /* descriptor */
        unsigned short  r_symbol;   /* symbol id */
        long            r_pos;      /* position */
};
```

Definitions for r_desc. The first group encodes the segment referenced, the second the size of relocation, and the last whether relocation is relative.

```
#define RD_TEXT    0x0000
#define RD_DATA    0x4000
#define RD_BSS     0x8000
```

```
#define RD_EXT      0xc000
#define RD_SEG      0xc000

#define RD_BYTE     0x0000
#define RD_WORD     0x1000
#define RD_LONG     0x2000
#define RD_SIZE     0x3000

#define RD_DISP     0x0800
```

For executable files, the short form (XR_RXEXEC) is used to allow a single module to be relocated. This enables the link editor to relocate a file to run at the different text and data base addresses required for different machines, or to convert it to pure text, fixed stack, etc, without recompiling.

```
struct xreloc {
        long  xr_cmd;
};
```

The first bit is set if the relocation references the text segment, the second if the relocation involves four bytes (as opposed to two). The last field is the position, or offset, within the current segment at which the relocation is to be performed.

```
#define XR_CODE   0x80000000    /* code/text segment */
#define XR_LONG   0x40000000    /* long/short operand */
#define XR_OFFS   0x3fffffff    /* 30 bit offset */
```

SEE ALSO

as(1S), ld(1S), nm(1S), strip(1S), brk(2), nlist(3)

NAME

acct - execution accounting file

SYNTAX

#include <sys/acct.h>

DESCRIPTION

Acct(2) causes entries to be made into an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

```
/*
 * Accounting structures
 */

typedef unsigned short comp_t;
        /* floating point format below: */
     /* 3 bits base 8 exp, 13 bits fraction */

struct  acct
{
        char    ac_comm[10]; /* Accounting command name */
        comp_t  ac_utime;    /* Accounting user time */
        comp_t  ac_stime;    /* Accounting system time */
        comp_t  ac_etime;    /* Accounting elapsed time */
        time_t  ac_btime;    /* Beginning time */
        short   ac_uid;      /* Accounting user ID */
        short   ac_gid;      /* Accounting group ID */
        short   ac_mem;      /* Average memory usage */
        comp_t  ac_io;       /* Number of disk IO blocks */
        dev_t   ac_tty;      /* Control typewriter */
        char    ac_flag;     /* Accounting flag */
};

extern  struct  acct  acctbuf;
extern  struct  inode *acctp; /* inode of accounting file */

#define AFORK   01    /* has executed fork, but no exec */
#define ASU     02    /* used super-user privileges */
```

If the process does an exec(2), the first 10 characters of the filename appear in ac comm. The accounting flag contains bits indicating whether exec(2) was ever accomplished, and whether the process ever had super-user privileges.

SEE ALSO

acct(2), sa(1)

NAME

ar - archive (library) file format

SYNTAX

#include <ar.h>

DESCRIPTION

The archive command ar is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor ld.

A file produced by ar has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARMAG   0177545
struct  ar_hdr {
        char    ar_name[14];
        long    ar_date;
        char    ar_uid;
        char    ar_gid;
        short   ar_mode;
        long    ar_size;
};
```

The name is a null-terminated string; the date is in the form of time(2); the user ID and group ID are numbers; the mode is a bit pattern per chmod(2); the size is counted in bytes.

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

SEE ALSO

ar(1S), ld(1S), nm(1S)

NOTES

Coding user and group IDs as characters is a botch.

NAME

core - format of core image file

DESCRIPTION

XENIX writes out a core image of a terminated process when any of various errors occur. See signal(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system's per-user data for the process, including the registers as they were at the time of the fault; see the system listings for the format of this area. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

In general the debugger adb(1) is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2)

NAME

dir - format of directories

SYNTAX

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry see, filsys(5). The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ  14
#endif
struct  direct
{
        ino_t   d_ino;
        char    d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases '..' has the same meaning as '.'.

SEE ALSO

filsys(5)

NAME

dump, ddate - incremental dump format

SYNTAX

```
#include <sys/types.h>
#include <sys/ino.h>
#include <dumprestor.h>
```

DESCRIPTION

Tapes used by dump and restor(1) contain:

A header record
Two groups of bit map records
A group of records describing directories
A group of records describing files

The format of the header record and of the first record of each description as given in the include file <dumprestor.h> is:

```
#define NTREC           20
#define MLEN            16
#define MSIZ            4096

#define TS_TAPE         1
#define TS_INODE        2
#define TS_BITS         3
#define TS_ADDR         4
#define TS_END          5
#define TS_CLRI         6
#define MAGIC           (int)60011
#define CHECKSUM        (int)84446
struct  spcl
{
        int     c_type;
        time_t  c_date;
        time_t  c_ddate;
        int     c_volume;
        daddr_t c_tapea;
        ino_t   c_inumber;
        int     c_magic;
        int     c_checksum;
        struct  dinode  c_dinode;
        int     c_count;
        char    c_addr[BSIZE];
} spcl;

struct  idates
{
        char    id_name[16];
        char    id_incno;
        time_t  id_ddate;
```

};

NTREC is the number of 512 byte records in a physical tape block. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the c_type field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE Tape volume label
TS_INODE
A file or directory follows. The c_dinode field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR A subrecord of a file description. See c_addr below.
TS_END End of tape record.
TS_CLRI A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC All header records have this number in c_magic.
CHECKSUM
Header records checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.
c_date The date the dump was taken.
c_ddate The date the file system was dumped from.
c_volume The current volume number of the dump.
c_tapea The current number of this (512-byte) record.
c_inumber
The number of the inode being dumped if this is of type TS_INODE.
c_magic This contains the value MAGIC above, truncated as needed.
c_checksum
This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode This is a copy of the inode as it appears on the file system; see filsys(5).
c_count The count of characters in c_addr.
c_addr An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not

sufficient space in this record to describe all of the blocks in a file, TS ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS END record and then the tapemark.

The structure idates describes an entry of the file /etc/ddate where dump history is kept. The fields of the structure are:

id_name The dumped filesystem is '/dev/id_nam'.
id_incno The level number of the dump tape; see dump(1).
id_ddate The date of the incremental dump in system format see types(5).

FILES

/etc/ddate

SEE ALSO

dump(1), dumpdir(1), restor(1), filsys(5), types(5)

NAME

environ - user environment

SYNTAX

extern char **environ;

DESCRIPTION

An array of strings called the 'environment' is made available by exec(2) when a process begins. By convention these strings have the form 'name=value'. The following names are used by various commands:

PATH The sequence of directory prefixes that sh, time, nice(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'. Login(1) sets PATH=:/bin:/usr/bin.

HOME A user's login directory, set by login(1) from the password file passwd(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as nroff or plot(1), which may exploit special terminal capabilities. See term(7) for a list of terminal types.

Further names may be placed in the environment by the export command and 'name=value' arguments in sh(1), or by exec(2). It is unwise to conflict with certain Shell variables that are frequently exported by '.profile' files: MAIL, PS1, PS2, IFS.

SEE ALSO

exec(2), sh(1), term(7), login(1)

NAME

filsys, fblk, ino - format of file system volume

SYNTAX

```
#include <sys/param.h>
#include <sys/fblk.h>
#include <sys/filsys.h>
#include <sys/ino.h>
```

DESCRIPTION

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the super block. The layout of the super block as defined by the include file <sys/filsys.h> is:

```
/*
 * Structure of the super-block
 */
struct  filsys {
    unsigned short s_isize; /* size in blocks of i-list */
    daddr_t s_fsize;        /* size in blocks of entire */
                            /* volume */
    short   s_nfree;        /* number of addresses */
                            /*   in s_free */
    daddr_t s_free[NICFREE];/* free block list */
    short   s_ninode;       /* number of i-nodes */
                            /*   in s_inode */
    ino_t   s_inode[NICINOD]; /* free i-node list */
    char    s_flock;        /* lock during free list */
                            /*   manipulation */
    char    s_ilock;        /* lock during i-list */
                            /*   manipulation */
    char    s_fmod;         /* super block modified flag */
    char    s_ronly;        /* mounted read-only flag */
    time_t  s_time;         /* last super block update */
    /* remainder not maintained by this version of system */
    daddr_t s_tfree;        /* total free blocks*/
    ino_t   s_tinode;       /* total free inodes */
    short   s_m;            /* interleave factor */
    short   s_n;            /* " " */
    char    s_fname[6];     /* file system name */
    char    s_fpack[6];     /* file system pack name */
    /* remainder is maintained for xenix */
    char    s_clean;        /* S_CLEAN if structure */
                            /*   is properly closed */
};

#define S_CLEAN 0106            /* arbitrary magic value  */
```

S_isize is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus the i-list is s_isize-2 blocks long. S_fsize is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The s_free array contains, in s_free[1], ... , s_free[s_nfree-1], up to NICFREE free block numbers. NICFREE is a configuration constant. S_free[0] is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file <sys/fblk.h> is:

```
struct fblk
{
        short   df_nfree;
        daddr_t df_free[NICFREE];
};
```

The fields df_nfree and df_free in a free block are used exactly like s_nfree and s_free in the super block. To allocate a block: decrement s_nfree, and the new block number is s_free[s_nfree]. If the new block address is 0, there are no blocks left, so give an error. If s_nfree became 0, read the new block into s_nfree and s_free. To free a block, check if s_nfree is NICFREE; if so, copy s_nfree and the s_free array into it, write it out, and set s_nfree to 0. In any event set s_free[s_nfree] to the freed block's address and increment s_nfree.

S_ninode is the number of free i-numbers in the s_inode array. To allocate an i-node: if s_ninode is greater than 0, decrement it and return s_inode[s_ninode]. If it was 0, read the i-list and place the numbers of all free inodes (up to NICINOD) into the s_inode array, then try again. To free an i-node, provided s_ninode is less than NICINODE, place its number into s_inode[s_ninode] and increment s_ninode. If s_ninode is already NICINODE, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

S_flock and s_ilock are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of s_fmod on disk is likewise

immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information. S_ronly is a write-protection indicator; its disk value is also immaterial.

S_time is the last time the super-block of the file system was changed. During a reboot, s_time of the super-block for the root file system is used to set the system's idea of the time.

The fields s_tfree, s_tinode, s_fname and s_fpack are not currently maintained.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-nodes are 64 bytes long, so 8 of them fit into a block. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node as given in the include file <sys/ino.h> is:

```
/*
 * Inode structure as it appears on
 * a disk block.
 */
struct dinode
{
        unsigned short  di_mode; /* mode and type of file */
        short   di_nlink;       /* number of links to file */
        short   di_uid;         /* owner's user id */
        short   di_gid;         /* owner's group id */
        off_t   di_size;        /* number of bytes in file */
        char    di_addr[40];    /* disk block addresses */
        time_t  di_atime;       /* time last accessed */
        time_t  di_mtime;       /* time last modified */
        time_t  di_ctime;       /* time created */
};
#define INOPB   8       /* 8 inodes per block */
/*
 * the 40 address bytes:
 *      39 used; 13 addresses
 *      of 3 bytes each.
 */
```

Di_mode tells the kind of file; it is encoded identically to the st_mode field of stat(2). Di_nlink is the number of directory entries (links) that refer to this i-node. Di_uid and di_gid are the owner's user and group IDs. Size is the number of bytes in the file. Di_atime and di_mtime are the times of last access and modification of the file contents (read, write or create) (see times(2)); Di_ctime records the time of last modification to the inode or to the file, and

is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the di addr field is occupied by the device code (see types(5)). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array di addr packed into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresses are singly, doubly, and triply indirect and point to blocks of 128 block pointers. Pointers in indirect blocks have the type daddr t (see types(5)).

For block b in a file to exist, it is not necessary that all blocks less than b exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

icheck(1), dcheck(1), shutdn(2), dir(5), mount(1M), stat(2), types(5), fsck(1M),

NAME

group - group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1), crypt(3), passwd(1), passwd(5)

NAME

mtab - mounted file system table

DESCRIPTION

Mtab resides in directory /etc and contains a table of devices mounted by the mount command. Umount removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last '/' is thrown away.

This table is present only so people can look at it. It does not matter to mount if there are duplicated entries nor to umount if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(1)

NAME

passwd - password file

DESCRIPTION

Passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
user's full name and optional information
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

Some programs depend on certain entries (such as "daemon" and "root"); these should not be removed.

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5), cron(8)

NAME

physio - physical i/o on raw devices

SYNTAX

#include <sys/param.h>

DESCRIPTION

Raw devices provide a means to avoid the operating system's normal block buffering mechanisms and to take advantage of special capabilities of various device controllers.

Raw physical i/o can be used with two classes of devices - devices with variable record sizes (such as an industry standard 9-track magnetic tape drive) and physically blocked devices with fixed record or sector sizes (such as any common disk drive). It is generally used for one of a small number of reasons.

Physical i/o improves transfer efficiency; i/o operations bypass the system block buffers, going directly to or from user memory. However, this is completely true only for those device controllers which fully support DMA (direct memory access) to all of memory. Manual pages in section 4 should be consulted for information about individual devices.

Physical i/o also provides more intimate control over devices than is possible with block-buffered i/o. If a floppy disk, for example, were to have an odd number of 256-byte sectors, the last sector could be accessed through physical i/o, but not through block-buffered i/o (see below). On standard magnetic tapes, as another example, physical i/o can be used to create or access records of arbitrary size; this is particularly important in dealing with tapes for use on non-XENIX systems.

Physical i/o operations to block or sector oriented devices must request transfer counts which are multiples of the system's block size, BSIZE, as defined in <sys/param.h>. Similarly, they must begin at device addresses which are a multiple of BSIZE. Any i/o request which does not comply with these restrictions will return an error status without initiating any i/o. To access a partial block at the end of a raw device, one must comply with these restrictions; the requested transfer count must be a multiple of BSIZE, but the system call will return the count of bytes actually transferred.

Because of variable record sizes on raw magnetic tape devices, the restrictions on raw i/o using disk devices do not apply to unformatted magnetic tape devices. However, because of this variable record length, lseek() calls to raw

magnetic tapes are quietly ignored.

SEE ALSO

lseek(2), read(2), write(2)

NOTES

On heavily loaded systems running certain implementations of XENIX, high overhead may be required to allocate contiguous regions of memory for processes requesting raw i/o.

Lseek() on raw magnetic tape should return an error indication. Also on raw magnetic tape, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

Uniform behavior at end of device has not yet been implemented for all device drivers.

Programs written for standard UNIX that do not follow these rules may require minor change to run under XENIX.

NAME

plot - graphics interface

DESCRIPTION

Files of this format are produced by routines described in plot(3), and are interpreted for various devices by commands described in plot(1). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in plot(3).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See plot(1).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first four bytes give the center of the circle, the next two the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in plot 4014 and plot ver.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of plot(1). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

```
4014        space(0, 0, 3120, 3120);
ver         space(0, 0, 2048, 2048);
300, 300s   space(0, 0, 4096, 4096);
450         space(0, 0, 4096, 4096);
```

SEE ALSO

plot(1), plot(3), graph(1)

NAME

termcap - terminal capability data base

SYNTAX

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, e.g., by vi(1) and curses(3). Terminals are described in termcap by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in termcap.

Entries in termcap consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed

dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give :ei=: if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default **^L**)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give :im=: if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by other function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of keypad transmit mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of other keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in keypad transmit mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on other function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with is)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)

ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't overstri
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
xb	bool		Beehive (f1=escape, f2=ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like **ce** \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the termcap file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200:\
        :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#
        :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:n
        :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in termcap are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has automatic margins (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a : in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines that deal with termcap use C strings, and strip the high bits of the output very late so that a **\200** comes out as a **\000** would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in termcap and to build up a description gradually, using partial descriptions with ex to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the termcap file to describe it or bugs in ex. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in /etc/termcap. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor.

Basic capabilities

The number of columns on each line for the terminal is given by the co numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the li capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it

should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the bs capability, unless a backspace is accomplished by a character other than ^H (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in termcap are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the termcap file usually assumes that this is on, i.e. **am**.

These capabilities suffice to describe hardcopy and glasstty terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with printf(3s) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the **%** encodings have the following meanings:

```
%d   as in printf, 0 origin
%2   like %2d
%3   like %3d
%.   like %c
%+x  adds x to value, then %.
%>xy if value > x adds y, no output.
%r   reverses order of line and column, no output
%i   increments line/column (for 1 origin)
%%   gives a single %
%n   exclusive or row and column with 0140 (DM2500)
%B   BCD (16*(x/10)) + (x%10), no output.
%D   Reverse coding (x-2*(x%16)), no output. (Delta Data).
```

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is cm=6\E&%r%2c%2Y. The Microterm ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, cm=^T%.%.. Terminals which use %. need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit \t, \n **^D** and \r, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cm=\E=%+ %+ .

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (nondestructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above

then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using termcap. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining - half bright is not usually an acceptable standout mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, this is acceptable, and although it may confuse some programs slightly, it can't be helped.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of ex, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9.** If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9.** If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the termcap 2 letter codes can be given in the **ko** capability, for example, :ko=cl,ll,sf,sb:, which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh.** It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the mime would be **:ma=^Kj^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than **^I** to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo

carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before if. This is useful where if is /usr/lib/tabset/std but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be last and the combined length of the two entries must not exceed 1024. Since termlib routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3), termcap(3), tset(1), vi(1), ul(1), more(1)

CREDIT

This utility was developed at the University of California at Berkeley and is used with permission.

NOTES

Ex allows only 256 characters for string capabilities, and the routines in termcap(3) do not check for overflow of this buffer. The total length of a single entry (excluding only

escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the vi program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

ttys - terminal initialization data

DESCRIPTION

The ttys file is read by the init program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to getty(8), which performs such tasks as baud-rate recognition, reading the login name, and calling login. For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (Getty will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, /dev.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1)

NAME

types - primitive system data types

SYNTAX

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in XENIX system code. Some data of these types are accessible to user code:

```
typedef long            daddr_t;        /* disk address */
typedef char *          caddr_t;        /* core address */
typedef unsigned short  ino_t;          /* i-node number */
typedef long            time_t;         /* a time */
typedef int             label_t[13];    /* program status */
typedef short           dev_t;          /* device code */
typedef long            off_t;          /* offset in file */

/* mmu dependent types */
typedef unsigned char  mloc_t;  /* memory region location */
typedef unsigned char  msize_t; /* memory region size */

/* selectors and constructor for device code */
#define major(x)        (short)(((unsigned)x>>8))
#define minor(x)        (short)(x&0377)
#define makedev(x,y)    (dev_t)((x)<<8|(y))
```

The form daddr_t is used for disk addresses except in an i-node on disk, see filsys(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The label_t variables are used to save the processor state while another process is running.

SEE ALSO

filsys(5), time(2), lseek(2), adb(1S)

NAME

utmp, wtmp - login records

SYNTAX

#include <utmp.h>

DESCRIPTION

The utmp file allows one to discover information about who is currently using XENIX. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
        char    ut_line[8];     /* tty name */
        char    ut_name[8];     /* user id */
        long    ut_time;        /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of time(2).

The wtmp file records all logins and logouts. Its format is exactly like utmp except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names '|' and '}' indicate the system-maintained time just before and just after a date command has changed the system's idea of the time.

Wtmp is maintained by login(1) and init(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by ac(1).

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

login(1), init(8), who(1), ac(1)

NAME

intro - introduction to games

DESCRIPTION

This section describes the recreational and educational programs found in the directory /usr/games. The availability of these programs may vary from system to system. A suggested procedure is to disallow their use during business hours by means of cron(1M).

NAME

arithmetic - provide drill in number facts

SYNTAX

/usr/games/arithmetic [**+-x/**] [range]

DESCRIPTION

Arithmetic types out simple arithmetic problems, and waits for an answer to be typed in. If the answer is correct, it types back "Right!", and a new problem. If the answer is wrong, it replies "What?", and waits for another answer. Every twenty problems, it publishes statistics on correctness and the time required to answer.

To quit the program, type an interrupt (delete).

The first optional argument determines the kind of problem to be generated; **+-x/** respectively cause addition, subtraction, multiplication, and division problems to be generated. One or more characters can be given; if more than one is given, the different types of problems will be mixed in random order; default is **+-**

Range is a decimal number; all addends, subtrahends, differences, multiplicands, divisors, and quotients will be less than or equal to the value of range. Default range is 10.

At the start, all numbers less than or equal to range are equally likely to appear. If the respondent makes a mistake, the numbers in the problem which was missed become more likely to reappear.

As a matter of educational philosophy, the program will not give correct answers, since the learner should, in principle, be able to calculate them. Thus the program is intended to provide drill for someone just past the first learning stage, not to teach number facts de novo. For almost all users, the relevant statistic should be time per problem, not percent correct.

NAME

backgammon - the game

SYNTAX

/usr/games/backgammon

DESCRIPTION

This program plays backgammon. It will ask whether you need instructions on how to play.

NAME

fortune, ching - the book of changes

SYNTAX

/usr/games/ching [hexagram]

/usr/games/fortune

DESCRIPTION

The I Ching or Book of Changes is an ancient Chinese oracle that has been in use for centuries as a source of wisdom and advice.

The text of the oracle (as it is sometimes known) consists of sixty-four hexagrams, each symbolized by a particular arrangement of six straight (---) and broken (- -) lines. These lines have values ranging from six through nine, with the even values indicating the broken lines.

Each hexagram consists of two major sections. The **Judgement** relates specifically to the matter at hand (E.g., "It furthers one to have somewhere to go.") while the **Image** describes the general attributes of the hexagram and how they apply to one's own life ("Thus the superior man makes himself strong and untiring.").

When any of the lines have the values six or nine, they are moving lines; for each there is an appended judgement which becomes significant. Furthermore, the moving lines are inherently unstable and change into their opposites; a second hexagram (and thus an additional judgement) is formed.

Normally, one consults the oracle by fixing the desired question firmly in mind and then casting a set of changes (lines) using yarrow-stalks or tossed coins. The resulting hexagram will be the answer to the question.

The XENIX oracle produces its answer on the standard output.

For those who wish to remain steadfast in the old traditions, the oracle will also accept the results of a personal divination using, for example, coins. To do this, cast the change and then type the resulting line values as an argument.

The impatient modern may prefer to settle for Chinese cookies; try fortune.

NAME

hangman - word guessing game

SYNTAX

/usr/games/hangman [dict]

/usr/games/words

DESCRIPTION

Hangman chooses a word at least seven letters long from a word list. The user is to guess letters one at a time.

The optional argument names an alternate word list.

FILES

/usr/dict/words the regular word list

DIAGNOSTICS

After each round, hangman reports the average number of guesses per round and the number of rounds.

SEE ALSO

words(6)

NAME

quiz - test your knowledge

SYNTAX

/usr/games/quiz [**-i** file] [**-t**] [category1 category2]

DESCRIPTION

Quiz gives associative knowledge tests on various subjects. It asks items chosen from category1 and expects answers from category2. If no categories are specified, quiz gives instructions and lists the available categories.

Quiz tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, quiz reports a score and terminates.

The **-t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The **-i** flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

```
line      = category newline | category ':' line
category  = alternate | category '|' alternate
alternate = empty | alternate primary
primary   = character | '[' category ']' | option
option    = '{' category '}'
```

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash '\' is used as with sh(1) to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, quiz will refrain from asking it.

FILES

/usr/games/quiz.k/*

NOTES

The construct 'a|ab' doesn't work in an information file. Use 'a{b}'.

NAME

words - word games

SYNTAX

/usr/games/words

DESCRIPTION

Words prints all the uncapitalized words in a word list that can be made from letters in string.

FILES

/usr/dict/words the regular word list

SEE ALSO

hangman(6)

NOTES

Hyphenated compounds are run together.

NAME

wump - the game of hunt-the-wumpus

SYNTAX

/usr/games/wump

DESCRIPTION

Wump plays the game of 'Hunt the Wumpus.' A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

NAME

intro - introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages and character set tables. Special text processing miscellany coming only with the XENIX Text Processing Package is designated **7T** .

NAME

ascii - map of ASCII character set

SYNTAX

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	|	175	}	176	~	177	del

FILES

/usr/pub/ascii

NAME

eqnchar - special character definitions for eqn

SYNTAX

eqn /usr/pub/eqnchar [files] | **troff** [options]

neqn /usr/pub/eqnchar [files] | **nroff** [options]

DESCRIPTION

Eqnchar contains troff and nroff character definitions for constructing characters that are not available on the Graphic Systems typesetter. These definitions are primarily intended for use with eqn and neqn.

FILES

/usr/pub/eqnchar

SEE ALSO

troff(1T), eqn(1T)

NAME

greek - graphics for the extended TTY-37 type-box

SYNTAX

cat /usr/pub/greek [| **greek** -Tterminal]

DESCRIPTION

Greek gives the mapping from ASCII to the "shift-out" graphics in effect between **SO** and **SI** on TELETYPE(Reg.) Model 37 terminals equipped with a 128-character type-box. These are the default greek characters produced by nroff(1). The filters of greek(1) attempt to print them on various other terminals. The file contains:

alpha	A	beta	B	gamma	\
GAMMA	G	delta	D	DELTA	W
epsilon	S	zeta	Q	eta	N
THETA	T	theta	O	lambda	L
LAMBDA	E	mu	M	nu	@
xi	X	pi	J	PI	P
rho	K	sigma	Y	SIGMA	R
tau	I	phi	U	PHI	F
psi	V	PSI	H	omega	C
OMEGA	Z	nabla	[	not	_
partial	]	integral	^		

FILES

/usr/pub/greek

SEE ALSO

troff(1T)

NAME

hier - file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative directory hierarchy.

```
/    root
/dev/
     devices (4)
     console
          main console, tty(4)
     tty* terminals, tty(4)
     fp*  floppy disks, fp, fp(4)
     rfp* raw floppy disks, hd(4)
     hd*  hard disks, hd, hd(4)
     rhd* raw hard disks, hd(4)
     ...
/bin/
     utility programs, cf /usr/bin/ (1)
     as   assembler first pass, cf /usr/lib/as2
     cc   C compiler executive, cf /usr/lib/c[012]
     ...
/lib/
     object libraries and other stuff, cf /usr/lib/
     libc.a
          system calls, standard I/O, etc. (2,3,3S)
     libm.a
          math routines (3M)
     libplot.a
          plotting routines, plot(3)
     ...
     as2  second pass of as(1)
     c[012]
          passes of cc(1)
     ...
/etc/
     essential data and dangerous maintenance utilities
     passwd
          password file, passwd(5)
     group
          group file, group(5)
     motd message of the day, login(1)
     mtab mounted file table, mtab(5)
     ddate
          dump history, dump(1)
     ttys properties of terminals, ttys(5)
     getty
          part of login, getty(8)
     init the father of all processes, init(8)
     rc   shell program to bring the system up
     cron the clock daemon, cron(8)
```

```
     mount
          mount(1)
     wall wall(1)
     ...
/tmp/
     temporary files, usually on a fast device, cf /usr/tmp/
     e*   used by ed(1)
     ctm* used by cc(1)
     ...
/usr/
     general-pupose directory, usually a mounted file system
     adm/ administrative information
          wtmp login history, utmp(5)
          messages
               hardware error messages messages(8)
/usr /bin
     utility programs, to keep /bin/ small
     tmp/ temporaries, to keep /tmp/ small
          stm* used by sort(1)
          raster
               used by plot(1)
     dict/
          word lists, etc.
          words
               principal word list, used by look(1)
          spellhist
               history file for spell(1)
     games/
          hangman
          quiz.k/
               what quiz(6) knows
               index
                    category index
               africa
                    countries and capitals
               ...
          ...
     include/
          standard #include files
          a.out.h
               object file layout, a.out(5)
          stdio.h
               standard I/O, stdio(3)
          math.h
               (3M)
          ...
          sys/ system-defined layouts, cf /usr/sys/h
               acct.h
                    process accounts, acct(5)
               buf.h
                    internal system buffers
               ...
```

```
          lib/ object libraries and stuff, to keep /lib/ small
               lint[12]
                    subprocesses for lint(1)
               llib-lc
                    dummy declarations for /lib/libc.a, used by
                    lint(1)
               llib-lm
                    dummy declarations for /lib/libc.m
               atrun
                    scheduler for at(1)
               struct/
                    passes of struct(1)
               ...
               tmac/
                    macros for troff(1)
                    tmac.an
                         macros for man(7)
                    tmac.s
                         macros for ms(7)
                    ...
               font/
                    fonts for troff(1)
                    R    Times Roman
                    B    Times Bold
                    ...
               uucp/
                    programs and data for uucp(1)
                    L.sys
                         remote system names and numbers
                    uucico
                         the real copy program
                    ...
               suftab
                    table of suffixes for hyphenation, used by
                    troff(1)
               units
                    conversion tables for units(1)
               eign list of English words to be ignored by ptx(1)
     /usr/     man/
          volume 1 of this manual, man(1)
               man0/
                    general
                    intro
                         introduction to volume 1, ms(7) format
                    xx   template for manual page
               man1/
                    chapter 1
                    as.1
                    mount.1m
                    ...
               cat1/
                    preprinted pages for man1/
```

```
          as.l
          mount.lm
     ...
spool/
     delayed execution files
     at/  used by at(1)
     lpd/ used by lpr(1)
          lock present when line printer is active
          cf*  copy of file to be printed, if necessary
          df*  daemon control file, lpd(8)
          tf*  transient control file, while lpr is
               working
     uucp/
          work files and staging area for uucp(1)
          LOGFILE
               summary log
          LOG.*
               log file for one transaction
mail/
     mailboxes for mail(1)
     uid  mail file for user uid
     uid.lock
          lock file while uid is receiving mail
wd   initial working directory of a user, typically wd
     is the user's login name
     .profile
          set environment for sh(1), environ(5)
     calendar
          user's datebook for calendar(1)
doc/ papers, mostly in volume 2 of this manual,
     typically in ms(7) format
     as/  assembler manual
     c    C manual
     ...
sys/ system source
     dev/ device drivers
          bio.c
               common code
          cat.c
               cat(4)
          dh.c DH11, tty(4)
          tty  tty(4)
          ...
     conf/
          hardware-dependent code
          mch.s
               assembly language portion
          conf configuration generator
          ...
     h/   header (include) files
          acct.h
               acct(5)
```

```
                    stat.h
                         stat(2)
                    ...
               sys/ source for system proper
                    main.c
                    pipe.c
                    sysent.c
                         system entry points
                    ...
/usr/     src/
     source programs for utilities, etc.
               cmd/ source of commands
                    as/  assembler
                         makefile
                              recipe for rebuilding the assembler
                         asl?.s
                              source of passl
                    ar.c source for ar(1)
                    ...
                    troff/
                         source for nroff and troff(1)
                         nmake
                              makefile for nroff
                         tmake
                              makefile for troff
                         font/
                              source for font tables,
                              /usr/lib/font/
                              ftR.c
                                   Roman
                              ...
                         term/
                              terminal characteristics tables,
                              /usr/lib/term/
                              tab300.c
                                   DASI 300
                              ...
                         ...
               libc/
                    source for functions in /lib/libc.a
                    crt/ C runtime support
                         ldiv.s
                              division into a long
                         lmul.s
                              multiplication to produce long
                         ...
                    csu/ startup and wrapup routines needed with
                         every C program
                         crt0.s
                              regular startup
                         mcrt0.s
                              modified startup for cc -p
```

```
                    sys/ system calls (2)
                         access.s
                         alarm.s
                         ...
                    stdio/
                         standard I/O functions (3S)
                         fgets.c
                         fopen.c
                         ...
                    gen/ other functions in (3)
                         abs.c
                         atof.c
                         ...
                    compall
                         shell procedure to compile libc
                    mklib
                         shell procedure to make /lib/libc.a
               libI77/
                    source for /lib/libI77
               libF77/
               ...
               games/
                    source for /usr/games
```

SEE ALSO

ls(1), ncheck(1), find(1), grep(1)

NOTES

The position of files is subject to change without notice.

NAME

man - macros to typeset manual

SYNTAX

nroff -man file ...

troff -man file ...

DESCRIPTION

These macros are used to lay out pages of this manual. A skeleton page may be found in the file /usr/man/man0/xx.

Any text argument t may be zero to six words. Quotes may be used to include blanks in a 'word'. If text is empty, the special treatment is applied to the next input line with text to be printed. In this way .I may be used to italicize a whole line, or .SM followed by .B to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents i are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by -man:

*R `(Reg)', trademark symbol in troff.

*S Change to default type size.

FILES

/usr/lib/tmac/tmac.an
/usr/man/man0/xx

SEE ALSO

troff(1T), man(1T)

NOTES

Relative indents don't nest.

REQUESTS

Request	Cause Break	If no Argument	Explanation
.B t	no	t=n.t.l.*	Text t is bold.
.BI t	no	t=n.t.l.	Join words of t alternating bold and italic.
.BR t	no	t=n.t.l.	Join words of t alternating bold and Roman.

.DT	no	.5i li...	Restore default tabs.
.HP i	yes	i=p.i.*	Set prevailing indent to i. Begin paragraph with hanging indent.
.I t	no	t=n.t.l.	Text t is italic.
.IB t	no	t=n.t.l.	Join words of t alternating italic and bold.
.IP x i	yes	x=""	Same as .TP with tag x.
.IR t	no	t=n.t.l.	Join words of t alternating italic and Roman.
.LP	yes	-	Same as .PP.
d.			
.PP	yes	-	Begin paragraph. Set prevailing indent to .5i.
.RE	yes	-	End of relative indent. Set prevailing indent to amount of starting .RS.
.RB t	no	t=n.t.l.	Join words of t alternating Roman and bold.
.RI t	no	t=n.t.l.	Join words of t alternating Roman and italic.
.RS i	yes	i=p.i.	Start relative indent, move left margin in distance i. Set prevailing indent to .5i for nested indents.
.SH t	yes	t=n.t.l.	Subhead.
.SM t	no	t=n.t.l.	Text t is small.
.TH n c x	yes	-	Begin page named n of chapter c; x is extra commentary, e.g. 'local', for page foot. Set prevailing indent and tabs to .5i.
.TP i	yes	i=p.i.	Set prevailing indent to i. Begin indented paragraph with hanging tag given by next text line. If tag doesn't fit, place it on separate line.

* n.t.l. = next text line; p.i. = prevailing indent

NAME

ms - macros for formatting manuscripts

SYNTAX

nroff -ms [options] file ...
troff -ms [options] file ...

DESCRIPTION

This package of nroff and troff macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through col(1).

The macro requests are defined below. Many nroff and troff requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first .PP:

.bp begin new page
.br break output line here
.sp n insert n spacing lines
.ls n (line spacing) n=1 single, n=2 double space
.na no alignment of right margin

Output of the eqn, neqn, refer, and tbl(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.s

SEE ALSO

eqn(1T), troff(1T), refer(1T), tbl(1T)

REQUESTS

Request	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AT	no	yes	Print 'Attached' and turn off line filling.
.AU x y	no	yes	Author's name follows. x is location and y is extension, ignored except in TM.
.B x	no	no	Print x in boldface; if no argument switch to boldface.
.B1	no	yes	Begin text to be enclosed in a box.
.B2	no	yes	End text to be boxed . print it.
.BT	date	no	Bottom title, automatically invoked at foot of page. May be redefined.

.BX x	no	no	Print x in a box.
.CS x...	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.CT	no	yes	Print 'Copies to' and enter no-fill mode.
.DA x	nroff	no	'Date line' at bottom of page is x. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS x	no	yes	Start of displayed text, to appear verbatim line-by-line. x=I for indented display (default), x=L for left-justified on the page, x=C for centered, x=B for make left-justified block, then center whole block. Implies .KS.
.EG	no	-	Print document in format for 'Engineer's Notes.' Must be first.
.EN	-	yes	Space after equation produced by eqn or neqn.
.EQ x y	-	yes	Precede equation; break out and add space. Equation number is y. The optional argument x may be I to indent equation (default), L to left-adjust the equation, or C to center the equation.
.FE	-	yes	End footnote.
.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.I x	no	no	Italicize x; if x missing, italic text follows.
.IM	no	no	Print document in format for an internal memorandum. Must be first.
.IP x y	no	yes	Start indented paragraph, with hanging tag x. Indentation is y ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MF	-	-	Print document in format for 'Memorandum for File.' Must be first.
.MR	-	-	Print document in format for 'Memorandum for Record.' Must be first.
.ND date	troff	no	Use date supplied (if any) only in special format positions; omit from page footer.
.NH n	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where n tells what level is wanted (default is 1).

.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.PT	pg #	-	Page title, automatically invoked at top of page. May be redefined.
.QE	-	yes	End quoted (indented and shorter) material.
.QP	-	yes	Begin single paragraph which is indented and shorter.
.QS	-	yes	Begin quoted (indented and shorter) material.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's are measured from current indentation.
.SG x	no	yes	Insert signature(s) of author(s), ignored except in TM. x is the reference line (initials of author and typist).
.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TA x...	5...	no	Set tabs in ens. Default is 5 10 15 ...
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TL	no	yes	Title follows.
.TM x...	no	-	Print document in technical memorandum format. Arguments are TM number, (quoted list of) case number(s), and file number. Must precede other requests.
.TR x	-	-	Print in BTL technical report format; report number is x. Must be first.
.TS x	-	yes	Begin table; if x is H table has repeated heading.
.UL x	-	no	Underline argument (even in troff).
.UX	-	no	'UNIX'; first time used, add footnote 'UNIX is a trademark of Bell Laboratories.'

NAME

terminals- conventional names

DESCRIPTION

These names are used by certain commands and are maintained as part of the shell environment (see sh(1),environ(5)).

trs16	TRS-80 Model II/Model 6 Console
adds25	TRS-80 DT-11 terminal in ADDS-25 mode
1620	DIABLO 1620 (and others using HyType II)
1620-12	same, in 12-pitch mode
300	DASI/DTC/GSI 300 (and others using HyType I)
300-12	same, in 12-pitch mode
300s	DASI/DTC 300/S
300s-12	same, in 12-pitch mode
33	TELETYPE(Reg.) Model 33
37	TELETYPE Model 37
40-2	TELETYPE Model 40/2
43	TELETYPE Model 43
450	DASI 450 (same as Diablo 1620)
450-12	same, in 12-pitch mode
450-12-8	same, in 12-pitch, 8 lines/inch mode
735	Texas Instruments TI735 (and TI725)
745	Texas Instruments TI745
dumb	terminals with no special features
hp	Hewlett-Packard HP264? series terminals
4014	Tektronix 4014
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
vt05	Digital Equipment Corp. VT05

Commands whose behavior may depend on the terminal accept arguments of the form **-Tterm,** where term is one of the names given above. If no such argument is present, a command may consult the shell environment for the terminal type.

SEE ALSO

stty(1), tabs(1), plot(1), sh(1), environ(5)
troff(1) for nroff

NOTES

The programs that ought to adhere to this nomenclature do so only fitfully.

NAME

intro - introduction to system maintenance procedures

DESCRIPTION

This section outlines certain procedures that will be of interest to those charged with the task of system maintenance. Included are discussions on such topics as boot procedures, recovery from crashes, file backups, etc.

NOTES

No manual can take the place of good, solid experience.

BOOT(8)

NAME

boot - how to bootstrap XENIX

DESCRIPTION

The term "boot" is used to describe the process of bringing XENIX up on your system. This should be distinguished from turning on the computer, or installing XENIX on your hard disk for the first time.

The first step in the boot procedure is to load the boot program. This step depends on the type of disk to boot from. To boot XENIX from a hard disk, press RESET. To boot XENIX from a system with only floppy disks, insert the boot disk in floppy drive 0 and press RESET. To boot XENIX from a floppy disk on a system that does have a hard disk, insert the boot disk in floppy drive 0, press RESET, and then press the BREAK key repeatedly until the boot message appears. If you don't press BREAK in this last procedure, the system will boot from the hard disk. When the boot program is loaded, the screen will clear and print the message:

```
Boot XENIX or TRSDOS? (x/t)
```

You should type "x <Enter>" for XENIX. Now the screen will read:

```
XENIX Boot
:
```

To boot XENIX, just press the <Enter> key. Your system will respond as if you had typed

```
xenix
```

As noted in the preceding chapter, if your system is shut down abnormally, the next time you attempt to "boot" the system XENIX will warn you that the last shutdown was abnormal. You should always respond "yes" when XENIX asks you whether to proceed with the cleaning of the file system. After XENIX has finished running the five-phase system-checking program called **fsck** and repaired system inconsistencies, the boot process will ordinarily continue normally. However, under certain error conditions discovered by **fsck** , the system may shut itself down and ask you to reboot. Repeat the normal boot procedure. When you have successfully booted XENIX on the hard disk, you will see some information about the size and release number of your system. This will be followed by the instruction:

```
Type Control-D to proceed with normal startup
(or give root password for system maintenance):
```

You should type Control-D at this point to bring up a working XENIX for yourself and other users. If you respond with the root password at this point, you will be positioned in "system maintenance" mode. In system maintenance mode, the functions of XENIX are limited; the occasions when even you, as system manager, will need to perform tasks in this mode are extremely rare; Be sure to type Control-D, unless you have a specific reason for being in system maintenance mode. After you type Control-D, you will be asked for the time. Although the system will accept <Enter> in lieu of typing the date, you are encouraged to give the system the full date. The accuracy of the dates and times stamped on all the files created and changed on XENIX depends on the time you enter. In actuality, the year (yy), month (mm), day (dd), and seconds (ss) are all optional; only the hour (hh) and minutes (mm) are required. The hour (hh) is in 24 hour format. After you enter the time, you will see the single word:

```
login:
```

Users may type their own login names, followed by their passwords, to begin working on the system. Note that the password will not appear on the screen as you type it. Once you have "logged in," the system will respond with a welcoming message, and you will see the XENIX prompt. If you are logged in as "root," this prompt will be a number sign (#). If you are logged in as an ordinary user, the prompt will be a dollar sign ($). You are now ready to begin using your XENIX system. One last thing you will undoubtedly want to know before going any further is how to stop XENIX safely. If you are logged in as root, you may type

```
# /etc/haltsys <Enter>
```

and you will see the message:

```
****NORMAL SYSTEM SHUTDOWN****
```

Other users may also shut down the system. They need to log out by typing Control-D and responding to the "login:" prompt by typing the "haltsys" after the login prompt:

```
login: haltsys
```

You may assign a password to the "haltsys" login, just as if haltsys were another user on the system, if you do not wish to give all your users the capacity to shut down the system.

FILES

/xenix	Default system to bootstrap
/z80ctl	Default I/O control program

NAME

cron - clock daemon

SYNTAX

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file /usr/lib/crontab. Since cron never exits, it should only be executed once. This is best done by running cron from the initialization process through the file /etc/rc; see init(8).

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by cron every minute.

The file /etc/passwd is used to determine the user and group id for the user "cron". If there is no such entry in the file then the user and group id will be that of the program that executes it; in particular, it will run as the super-user if executed from /etc/rc.

FILES

/usr/lib/crontab
/etc/passwd

NAME

getty - set terminal mode

SYNTAX

/etc/getty [char]

DESCRIPTION

Getty is invoked by init(8) immediately after a terminal is opened following a dial-up. It reads the user's login name and calls login(1) with the name as argument. While reading the name getty attempts to adapt the system to the speed and type of terminal being used.

Init calls getty with a single character argument taken from the ttys(5) file entry for the terminal line. This argument determines a sequence of line speeds through which getty cycles, and also the 'login:' greeting message, which can contain character sequences to put various kinds of terminals in useful states.

The user's name is terminated by a new-line or carriage-return character. In the second case CRMOD mode is set (see ioctl(2)).

The name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

If the terminal's 'break' key is depressed, getty cycles to the next speed appropriate to the type of line and prints the greeting message again.

Finally, login is called with the user's name as argument.

The following arguments from the ttys file are understood.

0 Cycles through 300-1200-150-110 baud. Useful as a default for dialup lines accessed by a variety of terminals.

- Intended for an on-line Teletype model 33, for example an operator's console.

1 Optimized for a 150-baud Teletype model 37.

2 Intended for an on-line 9600-baud terminal, for example the Textronix 4104.

3 Starts at 1200 baud, cycles to 300 and back. Useful with 212 datasets where most terminals run at 1200 speed.

5 Same as '3' but starts at 300.

4 Useful for on-line console DECwriter (LA36).

SEE ALSO

init(8), login(1), ioctl(2), ttys(5)

NAME

inir - root file system recovery during bootup

SYNTAX

/etc/inir

DESCRIPTION

Inir is invoked in place of /etc/init as the last step of the boot procedure when the root file structure was found to be flagged unclean. This will occur when /etc/shutdown or /etc/haltsys is not executed before halting and/or rebooting the system. Typically, this is the result of a hardware or software crash. Inir's role is to clean the root file system, and then either to invoke /etc/init or to halt the system so it can be rebooted normally.

When inir first is executed it prints

> The system was not shut down properly, and the
> root file system should be cleaned. Proceed (y/n)?

A 'n' reply will cause /etc/init to be invoked immediately. This is dangerous, and should only be done by those who are knowledgable or foolhardy. A 'y' reply should be entered, which will cause /bin/fsck to begin scanning the device '/dev/root' for damage.

If fsck finds that the root file structure was intact, /etc/inir will transfer control to /etc/init, and the boot will proceed normally. If the file structure was incorrect, fsck will repair it, and then call shutdn(2) to halt the system. You should then reboot. Since the root file system is now clean, the boot will proceed normally.

FILES

/dev/console, /etc/init, /bin/fsck

SEE ALSO

fsck(1m), mount(1m), shutdn(2), crash(8), init(8)

NAME

init, rc - process control initialization

SYNTAX

/etc/init
/etc/rc

DESCRIPTION

Init is invoked as the last step of the boot procedure (see boot(8)). Generally its role is to create a process for each typewriter on which a user may log in.

When init first is executed the console device, /dev/console, is opened for reading and writing and the shell is invoked immediately. This is used to bring up a single-user system. If the shell terminates, init comes up multi-user and the process described below is started.

When init comes up multiuser, it invokes a shell, with input taken from the file /etc/rc. This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then init reads the file /etc/ttys and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input, output and error files. Opening the typewriter will usually involve a delay, since the open is not completed until someone is dialed up and carrier established on the channel. Then /etc/getty is called with argument as specified by the last character of the ttys file line. Getty reads the user's name and invokes login(1) to log in the user and execute the shell.

Ultimately the shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of init, which has been waiting for such an event, wakes up and removes the appropriate entry from the file utmp, which records current users, and makes an entry in /usr/adm/wtmp, which maintains a history of logins and logouts. Then the appropriate typewriter is reopened and getty is reinvoked.

Init catches the hangup signal SIGHUP and interprets it to mean that the system should be brought from multi user to single user. Use 'kill -1 1' to send the hangup signal.

Init catches the interrupt signal SIGINT and interprets it to mean that the /etc/ttys file should be read again. The Shell process on each line which used to be active in ttys

but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the ttys file and sending an interrupt signal to the init process: use 'kill -2 1.' (Usually this is most easily accomplished via the enable(8) command).

Init catches the quit signal SIGQUIT and interprets it to mean that no more logins are allowed. Use 'kill -3 1' to send the quit signal.

FILES

/dev/tty*, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

SEE ALSO

login(1), kill(1), sh(1), ttys(5), getty(8)

NAME

lpd - line printer daemon

SYNTAX

/usr/lib/lpd

DESCRIPTION

Lpd is the daemon for the line printer. Lpd uses the directory /usr/spool/lpd. The file lock in that directory is used to prevent two daemons from becoming active. After the program has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with **df**. Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

L specifies that the remainder of the line is to be sent as a literal.

B specifies that the rest of the line is a file name.

F is the same as B except a form feed is prepended to the file.

U specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.

M is followed by a user ID; after the job is sent, a message is mailed to the user via the mail(1) command to verify the sending of the job.

Any error encountered will cause the daemon to wait and start over. This means that an improperly constructed **df** file may cause the same job to be submitted repeatedly.

Lpd is automatically initiated by the line printer command, lpr.

To restart lpd (in the case of hardware or software malfunction), it is necessary to first kill the old daemon (if still alive), and remove the lock file before initiating the new daemon. This is done automatically when the system is brought up, by /etc/rc, in case there were any jobs left in the spooling directory when the system last went down.

EXAMPLE

```
rm /usr/spool/lpd/lock ; /usr/lib/lpd
```

FILES

/usr/spool/lpd/* spool area for line printer daemon

/etc/passwd to get the user's name
/dev/lp line printer device

SEE ALSO

lpr(1)

NAME

makekey - generate encryption key

SYNTAX

/usr/lib/makekey

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e. to require a substantial fraction of a second).

The first eight input bytes (the input key) can be arbitrary ASCII characters. The last two (the salt) are best chosen from the set of digits, upper- and lower-case letters, and '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the output key.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

Makekey is intended for programs that perform encryption (e.g. ed and crypt(1)). Usually its input and output will be pipes.

SEE ALSO

crypt(1), ed(1)

NAME

messages - description of system console messages

DESCRIPTION

This section describes the various non-device system messages which may occur on the system console. Device-related messages start with the name of the device driver; these messages are documented by device in section 4.

Most of these system messages begin with "panic:" and are fatal (the system refuses to execute further). Fatal messages represent serious hardware problems or serious kernel software inconsistencies. Such internal inconsistencies are usually tracable to hardware problems themselves, often forms of memory failure.

A few message represent kernel operational probems, typically the overflow of a critical table. These potential problems are guarded against by the kernel, it takes such extreme situations to bring them about that they should never occur in normal system use.

The messages are presented below in alphabetical order. The accompanying text indicates fatal for those messages from which recovery is impossible. system inconsistency indicates a contradictory or impossible situation in the kernel, abnormal represents a probably legitimate but extreme situation, hardware indicates a clear hardware problem.

Many of these messages are accompanied by a device specification dev. This will print as nn/mm where nn is the major number and mm is the minor number of the offending device. If you do not recognize the device by its numbers type:

ls -l /dev | grep nn | grep mm

bad block on dev dev

A non-existent disk block was found on or is being inserted in the structure's free list. System inconsistency.

bad count on dev dev

Bad free count on dev dev

A structural inconsistency in the superblock of a file system. The system attempts a repair, but this message will probably be followed by more complaints about this file system. System inconsistency.

err on dev

This is the way that most device driver diagnostic messages start. The message will indicate the specific driver and complaint.

Inode table overflow

Each open file requires an inode entry to be kept in memory. When this table overflows the specific request is refused (usually open(2) or creat(2)). Although not fatal to the system, this event may damage the operation of various spoolers, deamons, the mailer, and other important utilities. Anomalous results and missing data files are a common result. If this message occurs during normal operation, if possible reconfigure your kernel with more inode table entries. Abnormal.

interrupt from unknown device, vec=xxxx

The CPU received an interrupt via a supposedly unused vector. This message is followed by "panic: unknown interrupt." Typically this event comes about when a hardware failure miscomputes the vector of a valid interrupt. Hardware.

no file

There are too many open files, the system has run out of entries in its "open file" table. The warnings given for the message "inode table overflow" apply here; if this occurs during normal operation use config(8) to increase the size of the file table. Abnormal.

no space on dev dev

This all-to-frequent message means that the specified file system has run out of free blocks. Although not normally as serious, the warnings discussed for "inode table overflow" apply: often programs are written casually and ignore the error code returned when they tried to write to the disk; this results in missing data and "holes" in data files. The system administrator should keep close watch on the amount of free disk space and take steps to avoid this situation.

** Normal System Shutdown **

This message appears when the system has been shutdown properly. It indicates that the machine may now be rebooted or powered down.

Out of inodes on dev dev.

The indicated file system has run out of free inodes. The number of inodes available on a file system is determined when mkfs(1) is run. The default number is quite generous, this message should be very rare. The only recourse is to remove some worthless files from that file system, or dump the entire system to a backup device, rerun mkfs(1) with more inodes specified, and restore the files from backup.

out of text
When programs linked with the -i or -n switch are run, a table entry is made so that only one copy of the pure text will be in memory even if there are multiple copies of the program running. This message appears when this table is full. The system refuses to run the program which caused the overflow. Note that there is only one entry in this table for each different pure text program, multiple copies of one program will not require multiple table entries. Each "sticky" program (see chmod(1)) requires a permanent entry in this table; non-sticky pure-text programs require an entry only when there is at least one copy being executed. See config(8) to increase the size of the text table.

panic: /0 trap
A divide-by-zero occured when executing kernel or device driver code. System inconsistency, fatal.

panic: blkdev

panic: devtab
An internal disk I/O request, already verified as valid, is discovered to be refering to a non-existant disk. System inconsistency, fatal.

panic: core free list
The internal memory allocation list has become corrupted. System inconsistency, fatal.

panic: free mm <1 pages

panic: freeing free mm
The internal memory management tables have become corrupted. System inconsistency, fatal.

panic: iaddress > 2^24
This indicates an attempted reference to an illegal block number (one so large that it could only occur on a file system larger than 8 billion bytes). System inconsistency, fatal.

panic: iinit
The super-block of the root file system could not be read. This message occurs only at boot time. Hardware, fatal.

panic: impossible data page

panic: impossible stack page

panic: impossible text page
The internal description of a task's memory has become corrupted. System inconsistency, fatal.

panic: Ill. TTY driver
An attempt was made to call an illegal tty driver. System inconsistency, fatal.

panic: Impossible trap type
The system hardware generated a trap of an unknown type. Hardware, fatal.

panic: IO err in swap
A fatal I/O error occurred while reading or writting the swap area. Hardware, fatal.

panic: Kernel data too large
An attempt to boot a XENIX kernel whose "data+bss" segments are too large. Rerun config(8) with reduced table and/or buffer requirements.

panic: mmblock

panic: mmufreemm

panic: mmugetmm

panic: mmumvmap

panic: mmuset:chk

panic: multi seg data
The internal memory management tables have become corrupted. System inconsistency, fatal.

panic: Multiplex Pipes not Present
An internal reference was made to multiplex pipes; this is an obsolete feature not present in this system. System inconsistency, fatal.

panic: no fs
A file system descriptor has disappeared from its table. System inconsistency, fatal.

panic: no imt
A mounted file system has disappeared from the mount table. System inconsistency, fatal.

panic: no procs
Each user is limited in the amount of simultaneous processes he can have; an attempt to create a new process when none is available or when the user's limit is

exceeded is refused. That is an occasional event and produces no console messages; this panic occurs when the kernel has certified that a free process table entry is available and yet can't find one when it goes to get it. System inconsistency, fatal.

panic: out of swap space
There is insufficient space on the swap disk to hold a task. The system refuses to create tasks when it feels there is insufficient disk space, but it is possible to create situations to fool this mechanism. Abnormal, fatal.

panic: overflow trap
The CPU generated an overflow trap while executing kernel or device driver code. System inconsistency, fatal.

panic: request for <1 mem
The internal memory management tables have become corrupted. System inconsistency, fatal.

panic: Running a dead proc
A dead process has found its way onto the "run" queue. System inconsistency, fatal.

panic: Sleeping on wchan 0
The kernel or a device driver has requested an illegal form of an operation called "sleep". System inconsistency, fatal.

panic: Text on Non-Sep
The internal description of a task's memory has become corrupted. System inconsistency, fatal.

panic: Timeout table overflow
The timeout table is full. Timeout requests are generated by device drivers, there should usually be room for one entry per system serial line plus ten more for other usages. This table can be increased via config(8). Abnormal, fatal.

panic: too much text
The internal description of a task's memory has become corrupted. System inconsistency, fatal.

panic: trap in sys
The CPU has generated an illegal instruction trap while executing kernel or device driver code. This message is preceeded with an information dump describing the trap. System inconsistency, fatal.

panic: unknown interrupt
The CPU received an interrupt via a supposedly unused vector. Typically this event comes about when a hardware failure miscomputes the vector of a valid interrupt. Hardware, fatal.

panic: zero wchan
A kernel or device driver has requested an illegal form of an operation called "sleep". System inconsistency, fatal.

Stray int: level n
The CPU received an interrupt via a supposedly unused vector. This message is followed by "panic: unknown interrupt." Typically this event comes about when a hardware failure miscomputes the vector of a valid interrupt. Hardware, fatal.

z80 panic: FD: Ill. Cmd
The z80 checks floppy disk commands for validity, so that a hardware or software problem does not cause a bad disk command to be executed by the z80. The probable cause of this error is an M68000 hardware problem. System inconsistency, fatal.

z80 panic: HD: Ill. Cmd
The z80 checks hard disk commands for validity, so that a hardware or software problem does not cause a garbage hard disk command to be executed by the z80. The probable cause of this error is an M68000 hardware problem. System inconsistency, fatal.

SEE ALSO
config(8)

NAME

update - periodically update the super block

SYNTAX

/etc/update

DESCRIPTION

Update is a program that executes the sync(2) primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

SEE ALSO

sync(2), sync(1), init(8)

NOTES

With update running, if the CPU is halted just as the sync is executed, a file system can be damaged. A fix would be to have sync(1) temporarily increment the system time by at least 30 seconds to trigger the execution of update. This would give 30 seconds grace to halt the CPU.

PERMUTED INDEX