UNISYS

System V

User's Reference Manual

Volume 1

October 1989

Printed in U S America
UP-15525
Update A

Priced Item

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Diablo is a registered trademark of Xerox Corporation.
Documenter's Workbench is a trademark of AT&T.
Hewlett-Packard is a registered trademark of Hewlett-Packard.
IBM is a registered trademark of International Business Machines Corporation
IBM S/370 is a trademark of International Business Machines Corporation.
TEKTRONIX is a trademark of TEKTRONIX Inc.
TELETYPE is a registered trademark of AT&T.
TRENDATA is a trademark of Trendata.
UNIX is a registered trademark of AT&T in the USA and other countries.
Versatec is a registered trademark of Versatec Corporation.
XENIX is a registered trademark of Microsoft Corporation.
Zerox is a trademark of Xerox Corporation.

About This Document

Purpose

This manual is intended to supplement information contained in the Administrator's Reference Manual and the Programmer's Reference Manual to provide an easy reference volume for those who must use a Unisys System V operating system.

Scope

This manual describes the user commands of the operating system.

Audience

The audience for this manual is users, programmers, analysts, and system support personnel.

Prerequisites

The user of this manual should be familiar with System V or another operating system derived from a UNIX® operating system.

UNIX is a registered trademark of AT&T in the USA and other countries.

This discussion provides the basic information you need to get started on your system: how to log in and log out, how to communicate through your terminal, and how to run a program. (See the User's Guide for a more complete introduction to the system.)

Logging In

You must connect to the operating system from a full-duplex ASCII terminal. You must also have a valid login id, which may be obtained (together with how to access your operating system) from the administrator of your system. Common terminal speeds are 120, 240, 480, 960, and 1920 characters per second (1200, 2400, 4800, 9600, and 19,200 baud). Some operating systems have different ways of accessing each available terminal speed, while other systems offer several speeds through a common access method. In the latter case, there is one preferred speed; if you access it from a terminal set to a different speed, you are greeted by a string of meaningless characters (the **login:** message at the wrong speed). Keep hitting the break, interrupt, or attention key until the **login:** message appears.

Most terminals have a speed switch that should be set to the appropriate speed and a half-/full-duplex switch that should be set to full-duplex. When a connection has been established, the system types **login:**. You respond by typing your login id followed by the return key. If you have a password, the system asks for it but will not print, or echo, it on the terminal. After you have logged in, the return, new-line, and line-feed keys all have equivalent meanings.

Make sure you type your login name in lowercase letters. Typing uppercase letters causes the operating system to assume that your terminal can generate only uppercase letters and treats all letters as uppercase for the remainder of your login session. The shell prints a $ on your screen when you have logged in successfully.

When you log in, a message-of-the-day may greet you before you receive your prompt. For more information, consult *login*(1), which discusses the login sequence in more detail, and *stty*(1), which tells you how to describe your terminal to the system. *profile*(4) (in the Programmer's Reference Manual) explains how to accomplish this last task automatically every time you log in.

Logging Out

There are two ways to log out:

- If you've dialed in, you can simply hang up the phone.
- You can log out by typing an end-of-file indication (ASCII EOT character, usually typed as < **Ctrl** > **D**) to the shell. The shell terminates, and the **login:** message appears.

How to Communicate Through Your Terminal

When you type to the operating system, your individual characters are being gathered and temporarily saved. Although they are echoed back to you, these characters will not be given to a program until you type a return (or new-line) as described above.

Operating system terminal input/output is full duplex. It has full read-ahead, which means that you can type at any time, even while a program is being executed. Of course, if you type during output, your input characters have output characters interspersed among them. In any case, whatever you type is saved and interpreted in the correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded.

The character @ cancels all the characters typed before it on a line, effectively deleting the line. (@ is called the line kill character.) The character # erases the last character typed. Successive uses of # erases characters back to, but not beyond, the beginning of the line; @ and # can be typed as themselves by preceding them with \ (thus, to erase a \, you need two #s). These default erase and line kill characters can be changed; see *stty*(1).

< **Ctrl** > **S** (also known as the ASCII **DC3** character) is typed by pressing the control key and the alphabetic s simultaneously and is used to stop output temporarily. It is useful with CRT terminals to prevent output from disappearing before it can be read. Output is resumed when a < **Ctrl** > **Q** (also known as **DC1**) is typed. Thus, if you had typed **cat yourfile** and the contents of **yourfile** were passing by on the screen more rapidly than you could read it, you would type < **Ctrl** > **S** to freeze the output for a moment. Typing < **Ctrl** > **Q** allows the output to resume its rapid pace. The < **Ctrl** > **S** and < **Ctrl** > **Q** characters are not passed to any other program when used in this manner.

The ASCII < **Del** > (a.k.a. rubout) character is not passed to programs but instead generates an *interrupt signal,* just like the break, interrupt, or attention signal. This signal generally causes whatever program you are running to terminate. It is typically used to stop a long printout that you do not want. Programs, however, can arrange either to ignore this signal altogether or to be notified and take a specific action when it happens (instead of being terminated). The editor *ed*(1), for example, catches interrupts and stops what *it* is doing, instead of terminating, so an interrupt can be used to halt an editor printout without losing the file being edited.

Besides adapting to the speed of the terminal, the operating system tries to be intelligent as to whether you have a terminal with the new-line function, or whether it must be simulated with a carriage-return and line-feed pair. In the latter case, all *input* carriage-return characters are changed to line-feed characters (the standard line delimiter), and a carriage-return and line-feed pair is echoed to the terminal. If you get into the wrong mode, the *stty*(1) command rescues you.

< **Tab** > characters are used freely in operating system source programs. If your terminal does not have the tab function, you can arrange to have tab characters changed into spaces during output, and echoed as spaces during input. Again, the *stty*(1) command sets or resets this mode. The system assumes that tabs are set every eight character positions. The *tabs*(1) command sets tab stops on your terminal, if that is possible.

How to Run a Program

When you have successfully logged into the operating system, a program called the shell is communicating with your terminal. The shell reads each line you type, splits the line into a command name and its arguments, and executes the command. A command is simply an executable program. Normally, the shell looks first in your current directory (see "The Current Directory" that follows) for the named program and, if none is there, then in system directories, such as **/bin** and **/usr/bin**. There is nothing special about system-provided commands except that they are kept in directories where the shell can find them. You can also keep commands in your own directories and instruct the shell to find them there. See the manual entry for *sh*(1), under the sub-heading Parameter Substitution, for the discussion of the **$PATH** shell environment variable.

The command name is the first word on an input line to the shell; the command and its arguments are separated from one another by space or tab characters.

When a program terminates, the shell ordinarily regains control and give you back your prompt to indicate that it is ready for another command. The shell has many other capabilities, which are described in detail in *sh*(1).

The Current Directory

The operating system has a file system arranged in a hierarchy of directories. When you received your login id, the system administrator also created a directory for you (ordinarily with the same name as your login id, and known as your *login* or *home* directory). When you log in, that directory becomes your *current* or *working* directory, and any file name you type is, by default, assumed to be in that directory. Because you are the owner of this directory, you have full permissions to read, write, alter, or remove its contents. Permissions to enter or modify other directories and files have been granted or denied to you by their respective owners or by the system administrator. To change the current directory, use *cd*(1).

Pathnames

To refer to files or directories not in the current directory, you must use a pathname. Full pathnames begin with /, which is the name of the *root* directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory (followed by a /), until finally the file or directory name is reached (e.g., /usr/ae/filex refers to file **filex** in directory **ae**, while **ae** is itself a subdirectory of **usr**, and **usr** is a subdirectory of the root directory). Use *pwd*(1) to print the full pathname of the directory you are working in. See *intro*(2) in the Programmer's Reference Manual for a formal definition of *pathname*.

If your current directory contains subdirectories, the pathnames of their respective files begin with the name of the corresponding subdirectory (*without* a prefixed /). A pathname may be used anywhere a file name is required.

Important commands that affect files are *cp*(1), *mv* (see *cp*(1)), and *rm*(1), which respectively copy, move (i.e., rename), and remove files. To find out the status of files or directories, use *ls*(1). Use *mkdir*(1) for making directories and *rmdir* (see *rm*(1)) for removing them.

Text Entry and Display

Almost all text is entered through an editor. Common examples of operating system editors are *ed*(1) and *vi*(1). The commands most often used to print text on a terminal are *cat*(1), *pr*(1), and *pg*(1). The *cat*(1) command displays the contents of ASCII text files on the terminal, with no processing at all. The *pr*(1) command paginates the text, supplies headings, and has a facility for multi-column output. The *pg*(1) command displays text in successive portions no larger than your terminal screen.

Writing a Program

Once you have entered the text of your program into a file with an editor, you are ready to give the file to the appropriate language processor. The processor accepts only files observing the correct naming conventions: all C programs must end with the suffix **.c**, and FORTRAN programs must end with **.f**. The output of the language processor is left in a file named **a.out** in the current directory, unless you have invoked an option to save it in another file. (Use *mv*(1) to rename **a.out**.) If the program is written in assembly language, you may need to load library subroutines with it (see *ld*(1) in the Programmer's Reference Manual).

When you have completed this process without provoking any diagnostics, you may run the program by giving its name to the shell in response to the $ prompt. Your programs can receive arguments from the command line just as system programs do; see *exec*(2) in the Programmer's Reference Manual. For more information on writing and running programs, see the Programmer's Guide.

Communicating with Others

Certain commands provide *inter-user* communication. Even if you do not plan to use them, it would be well to learn something about them because someone else may try to contact you. *mail*(1) or *mailx*(1) leave a message whose presence is announced to another user when he or she next logs in and at periodic intervals during the session. To communicate with another user currently logged in, *write*(1) is used. The corresponding entries in this manual also suggest how to respond to these two commands if you are their target.

See the tutorials in the User's Guide for more information on communicating with others.

How to Use This Document

The commands appear as follows:

1. Commands and Application Programs

The section begins with a page labelled *intro*. Entries following the *intro* page are arranged alphabetically and may consist of more than one page. Some entries describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its primary name. An example of such an entry is *chown*(1), which also describes the *chgrp* command.

References with numbers other than those mentioned previously are contained in the appropriate section of another manual. References with a (1M) or (7) following the command name mean that the man page is contained in the Administrator's Reference Manual. References with a (1) or (1G) can be found in either the User's Reference Manual or the Programmer's Reference Manual. All other referenced commands including (2), (3), (3C), (3X), (3M), (3S), (3W), (3N), (4) and (5) can be found in the Programmer's Reference Manual.

Some man pages are unique to the International Enhancements (IE) utility and some have been modified to work with the IE utility. In both instances, these utilities are shown in the Table of Contents with the letters IE following their name.

Organization

This manual contains the following sections:

Permuted Index

This index is derived from the Table of Contents (which lists both primary and secondary command entries and gives an abstract of each command). The Permuted Index is used by searching the middle column for a key word or phrase. The right column contains the name of the manual page that contains the command. The left column contains additional useful information about the command.

Section 1. Commands and Application Programs

The entries in Section 1 describe programs intended to be invoked directly by the user or by command language procedures, as opposed to subroutines, which are called by the user's programs. These include general utility commands, commands used in communicating with other systems, and commands used for graphics and computer-aided design. Commands generally reside in the directory **/bin** (for **bin**ary programs). In addition, some programs reside in **/usr/bin.** These directories are searched automatically by the command interpreter called the *shell*. UNIX systems running on Unisys computers also have a directory called **/usr/lbin,** containing local commands.

All entries are presented using the following format (though some of these headings might not appear in every entry):

- **NAME** gives the primary name (and secondary name(s), as the case may be) and briefly states its purpose.
- **SYNOPSIS** summarizes the usage of the program being described. A few explanatory conventions are used, particularly in the **SYNOPSIS:**
  - **Boldface** strings are literals and are to be typed just as they appear.
  - *Italic* strings usually represent substitutable argument prototypes and command names found elsewhere in the manual.
  - Square brackets [ ] around an argument prototype indicate that the argument is optional. When an argument prototype is given as name or file, 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. An argument beginning with a minus (-), plus (+), or an equal sign (=) is often taken to be some sort of flag 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 -, +, or =.
- **DESCRIPTION** provides an overview of the utility. The **Internationalization** sub-section includes modifications that allow it to work with the International Enhancements utility.

- **EXAMPLE(S)** gives example(s) of usage, where appropriate.
- **FILES** contains the file names that are referenced by the program.
- **EXIT CODES** discusses values set when the command terminates. The value set is available in the shell environment variable "?" (see *sh*(1)).
- **RETURN VALUES** identifies values which are returned during the execution of a command and/or program.
- **NOTES and CAVEATS** gives information that may be helpful under the particular circumstances described.
- **SEE ALSO** offers pointers to related information.
- **DIAGNOSTICS** discusses the error messages that may be produced. Messages that are intended to be self-explanatory are not listed.
- **WARNINGS** discusses the limits or boundaries of the respective commands.
- **BUGS** lists known faults in software that have not been rectified. Occasionally, a suggested short-term remedy is also described.
- **RESTRICTIONS** provides known restrictions and deficiencies. Occasionally, the suggested fix is also described.
- **SUPPORT STATUS** points out commands that may not be supported in future releases.

Related Product Information

Programmer's Reference Manual
Administrator's Reference Manual
User's Guide

Contents

(The following are contained in two volumes.)

1. Commands and Application Programs

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

IE: Indicates commands that are either unique to the International Enhancements (IE) utility or that have been modified to work with the IE utility.

Permuted Index

NAME

intro - introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, commands available for your computer. The commands in this section should be used along with those listed in Sections 1, 2, 3, 4, and 5 of the Programmer's Reference Manual. References of the form *name*(1), *name*(2), *name*(3), *name*(4), and *name*(5) refer to entries in the Programmer's Reference Manual. References of the form *name*(1M), *name*(7) refer to entries in the Administrator's Reference Manual. References of the form *name*(1), *name*(1C), *name*(1G) refer to entries in this manual. Certain distinctions of purpose are made in the headings.

The following Utility packages are delivered with the computer:

Base System
Editing Package
Extended Terminal Interface
Crypt Utilities Package
2 Kilobyte File System Utility Package
Network Support Utilities Package
Remote File Sharing Utilities Package

Command Syntax Standard: Rules

These command syntax rules are not followed by all current commands, but all new commands use them. The *getopts*(1) command should be used by all shell procedures to parse positional parameters and to check for legal options. It supports Rules 3-10 in the following list. The enforcement of the other rules must be done by the command itself.

1. Command names (*name* as described previously) must be between two and nine characters long.
2. Command names must include only lowercase letters and digits.
3. Option names (*option* as described previously) must be one character long.
4. All options must be preceded by a "-".
5. Options with no arguments may be grouped after a single -.
6. The first option-argument (*optarg* as described previously) following an option must be preceded by white space.

7. Option-arguments cannot be optional.
8. Groups of option-arguments following an option must either be separated by commas or separated by white space and quoted (e.g., `-o xxx,z,yy` or `-o "xxx z yy"`).
9. All options must precede operands (*cmdarg* as described previously) on the command line.
10. -- may be used to indicate the end of the options.
11. The order of the options relative to one another should not matter.
12. The relative order of the operands (*cmdarg* as described previously) may affect their significance in ways determined by the command with which they appear.
13. - preceded and followed by white space should only be used to mean standard input.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system and giving the cause for termination and (in the case of normal termination) one supplied by the program [see *wait*(2) and *exit*(2)]. The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and non-zero to indicate troubles such as erroneous parameters or bad or inaccessible data. It is called variously exit code, exit status, or return code and is described only where special conventions are involved.

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

NAME

300, 300s - handle special functions of DASI 300 and 300s terminals

SYNOPSIS

300 [+12] [-n] [-dt,l,c]

300s [+12] [-n] [-dt,l,c]

DESCRIPTION

The *300* command supports special functions and optimizes the use of the DASI 300 (GSI 300 or DTC 300) terminal; *300s* performs the same functions for the DASI 300s (GSI 300s or DTC 300s) terminal. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. In the following discussion of the *300* command, it should be noted that unless your system contains the DOCUMENTER'S WORKBENCH Software, references to certain commands (e.g., *nroff*, *neqn*, *eqn,* etc.) will not work. It also attempts to draw Greek letters and other special symbols. It permits convenient use of 12-pitch text. It also reduces printing time 5 to 70 percent. The *300* command can be used to print equations neatly, in the sequence:

neqn file ... | nroff | 300

WARNING: If your terminal has a PLOT switch, make sure it is turned on before *300* is used.

The behavior of *300* can be modified by the optional flag arguments to handle 12-pitch text, fractional line spacings, messages, and delays.

+12 permits use of 12-pitch, 6 lines per inch text. DASI 300 terminals normally allow only two combinations, 10-pitch, 6 lines per inch, or 12-pitch, 8 lines per inch. To obtain the 12-pitch, 6 lines per inch combination, the user should turn the PITCH switch to 12, and use the +12 option.

-n controls the size of half-line spacing. A half-line is, by default, equal to four vertical plot increments. Because each increment equals 1/48 of an inch, a 10-pitch line-feed requires eight increments, while a 12-pitch line-feed needs only six. The first digit of *n* overrides the default value, thus allowing for individual taste in the appearance of subscripts and superscripts.

-d*t, l, c* controls delay factors. The default setting is **-d3,90,30**. DASI 300 terminals sometimes produce peculiar output when faced with very long lines, too many tab characters, or long strings of blankless, non-identical characters. One null (delay) character is inserted in a line

for every set of *t* tabs, and for every contiguous string of *c* non-blank, non-tab characters. If a line is longer than *l* bytes, 1+(total length)/20, nulls are inserted at the end of that line. Items can be omitted from the end of the list, implying use of the default values. Also, a value of zero for *t* (*c*) results in two null bytes per tab (character). The former may be needed for C programs, the latter for files like **/etc/passwd**. Because terminal behavior varies according to the specific characters printed and the load on a system, the user may have to experiment with these values to get correct output. The **-d** option exists only as a last resort for those few cases that do not otherwise print properly. For example, the file **/etc/passwd** may be printed using **-d3,30,5**. The value **-d0,1** is a good one to use for C programs that have many levels of indentation.

Note that the delay control interacts heavily with the prevailing carriage return and line-feed delays. The *stty*(1) modes **nl0 cr2** or **nl0 cr3** are recommended for most uses.

The *300* command can be used with the *nroff* **-s** flag or **.rd** requests, when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the return key in these cases, you must use the line-feed key to get a response.

In many cases, the following sequences are equivalent:

nroff -T300 files ... and nroff files ... ¦ 300
nroff -T300-12 files ... and nroff files ... ¦ 300 + 12

The use of *300* can thus often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of *300* may produce better aligned output.

SEE ALSO

450(1), mesg(1), graph(1G), stty(1), tabs(1), tplot(1G), greek(1)

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.

If your output contains Greek and/or reverse line-feeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, which distorts Greek characters and misaligns the first line of text after one or more reverse line-feeds.

NAME

4014 - paginator for the TEKTRONIX 4014 terminal

SYNOPSIS

4014 [-t] [-n] [-cN] [-pL] [file]

DESCRIPTION

The output of *4014* is intended for a TEKTRONIX 4014 terminal; *4014* arranges for 66 lines to fit on the screen, divides the screen into *N* columns, and contributes an eight-space page offset in the (default) single-column case. Tabs, spaces, and backspaces are collected and plotted when necessary. TELETYPE Model 37 half and reverse line sequences are interpreted and plotted. At the end of each page, *4014* waits for a newline (empty line) from the keyboard before continuing on to the next page. In this wait state, the command !*cmd* will send the *cmd* to the shell.

The command line options are:

-t Do not wait between pages (useful for directing output into a file).

-n Start printing at the current cursor position and never erase the screen.

-c*N* Divide the screen into *N* columns and wait after the last column.

-p*L* Set page length to *L*; the *L* accepts the scale factors **i** (inches) and **l** (lines); default is lines.

SEE ALSO

pr(1), tc(1)

[This page left blank.]

NAME

450 - handle special functions of the DASI 450 terminal

SYNOPSIS

450

DESCRIPTION

The *450* command supports special functions of, and optimizes the use of, the DASI 450 terminal, or any terminal that is functionally identical, such as the Diablo 1620 or Xerox 1700. It converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. It also attempts to draw Greek letters and other special symbols in the same manner as *300*(1). It should be noted that, unless your system contains DOCUMENTER'S WORKBENCH Software, certain commands (e.g., *eqn*, *nroff*, *tbl*, etc.) will not work. Use *450* to print equations neatly, in the sequence:

neqn file ... nroff 450

WARNING: Make sure that the PLOT switch on your terminal is on before *450* is used. The SPACING switch should be put in the desired position (either 10- or 12-pitch). In either case, vertical spacing is 6 lines per inch, unless dynamically changed to 8 lines per inch by an appropriate escape sequence.

Use *450* with the *nroff* **-s** flag or **.rd** requests when it is necessary to insert paper manually or change fonts in the middle of a document. Instead of hitting the return key in these cases, you must use the line-feed key to get a response.

In many cases, the use of *450* can be eliminated in favor of one of the following:

nroff -T450 files ...

or

nroff -T450-12 files ...

The use of *450* can often be avoided unless special delays or options are required; in a few cases, however, the additional movement optimization of *450* may produce better aligned output.

SEE ALSO

300(1), mesg(1), stty(1), tabs(1), graph(1G), tplot(1G), greek(1)

BUGS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.

If your output contains Greek and/or reverse line-feeds, use a friction-feed platen instead of a forms tractor; although good enough for drafts, the latter has a tendency to slip when reversing direction, which distorts Greek characters and misaligns the first line of text after one or more reverse line-feeds.

NAME

acctcom - search and print process accounting file(s)

SYNOPSIS

acctcom [[options] [file]] ...

DESCRIPTION

The *acctcom* command reads *file*, the standard input, or **/usr/adm/pacct**, in the form described by *acct*(4), and writes selected records to the standard output. Each record represents the execution of one process. The output shows the **COMMAND NAME, USER, TTYNAME, START TIME, END TIME, REAL (SEC), CPU (SEC), MEAN SIZE(K)**, and optionally, **F** (the *fork/exec* flag: **1** for *fork* without *exec*), **STAT** (the system exit status), **HOG FACTOR, KCORE MIN, CPU FACTOR, CHARS TRNSFD**, and **BLOCKS READ** (total blocks read and written).

The command name is prepended with a # if it was executed with superuser privileges. If a process is not associated with a known terminal, a ? is printed in the **TTYNAME** field.

If no *files* are specified, and if the standard input is associated with a terminal or **/dev/null** (as is the case when using & in the shell), **/usr/adm/pacct** is read; otherwise the standard input is read.

If any *file* arguments are given, they are read in their respective order. Each file is normally read forward; that is, in chronological order by process completion time. The file **/usr/adm/pacct** is usually the current file to be examined; a busy system may need several such files of which all but the current file are found in **/usr/adm/pacct?**. The options are:

-a Show some average statistics about the processes selected. The statistics will be printed after the output records.

-b Read backwards, showing latest commands first. This option has no effect when the standard input is read.

-f Print the *fork/exec* flag and system exit status columns in the output.

-h Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution. This "hog factor" is computed as:

(total CPU time)/(elapsed time).

-i	Print columns containing the I/O counts in the output.
-k	Instead of memory size, show total kcore-minutes.
-m	Show mean core size (the default).
-r	Show CPU factor (user time)/(system-time + user-time).
-t	Show separate system and user CPU times.
-v	Exclude column headings from the output.
-l *line*	Show only processes belonging to terminal **/dev/***line*.
-u *user*	Show only processes belonging to *user* that may be specified by: a user ID, a login name that is then converted to a user ID, a **#**, which designates only those processes executed with super-user privileges, or **?**, which designates only those processes associated with unknown user IDs.
-g *group*	Show only processes belonging to *group*. The *group* may be designated by either the group ID or group name.
-s *time*	Select processes existing at or after *time*, given in the format *hr* [*:min* [*:sec*]].
-e *time*	Select processes existing at or before *time*.
-S *time*	Select processes starting at or after *time*.
-E *time*	Select processes ending at or before *time*. Using the same *time* for both **-S** and **-E** shows the processes that existed at *time*.
-n *pattern*	Show only commands matching *pattern* that may be a regular expression as in *ed*(1) except that + means one or more occurrences.
-q	Do not print any output records, just print the average statistics as with the **-a** option.
-o *ofile*	Copy selected process records in the input data format to *ofile*; suppress standard output printing.
-H *factor*	Show only processes that exceed *factor*, where factor is the "hog factor" as explained in option **-h** above.
-O *sec*	Show only processes with CPU system time exceeding *sec* seconds.

-C *sec* Show only processes with total CPU time, system plus user, exceeding *sec* seconds.

-I *chars* Show only processes transferring more characters than *chars*.

Listing options together has the effect of a logical AND.

FILES

/etc/passwd
/usr/adm/pacct
/etc/group

SEE ALSO

ps(1)
acct(1M), acctcms(1M), acctcon(1M), acctmerg(1M), acctprc(1M), acctsh(1M), runacct(1M), su(1M) in the Administrator's Reference Manual
acct(2), acct(4), utmp(4) in the Programmer's Reference Manual

BUGS

The *acctcom* command only reports on processes that have terminated; use *ps*(1) for active processes. If *time* exceeds the present time, *time* is interpreted as occurring on the previous day.

[This page left blank.]

NAME

asa - interpret ASA carriage control characters

SYNOPSIS

asa [files]

DESCRIPTION

The *asa* command interprets the output of FORTRAN programs that utilize ASA carriage control characters. It processes either the *files* whose names are given as arguments or the standard input if no file names are supplied. The first character of each line is assumed to be a control character; their meanings are:

' ' (blank) single new line before printing

0 double new line before printing

1 new page before printing

\+ overprint previous line.

Lines beginning with other than the above characters are treated as if they began with ' '. The first character of a line is *not* printed. If any such lines appear, an appropriate diagnostic will appear on standard error. This program forces the first line of each input file to start on a new page.

To view correctly the output of FORTRAN programs which use ASA carriage control characters, *asa* could be used as a filter as follows:

```
a.out | asa | lp
```

and the output, properly formatted and paginated, would be directed to the line printer. FORTRAN output sent to a file could be viewed by:

```
asa file
```

[This page left blank.]

NAME

assist - assistance using System V commands

SYNOPSIS

assist [-s] [-c [name]]

DESCRIPTION

The *assist* command invokes the ASSIST menu interface software for the System V. The ASSIST menus categorize System V commands according to function in a hierarchy. The menus lead to full-screen forms that aid you in the execution of a syntactically correct System V command line.

If you type *assist* without options, you enter at the top of the menu interface hierarchy. New users may select an introductory tutorial explaining how to use the ASSIST software.

There are two options for ASSIST:

-c *name* This option invokes a command form in the user's current directory. *Name* is an ASSIST-supported System V command or the name of a walkthrough.

-s This option reinvokes the ASSIST setup module to check/modify your terminal variable. You can also access the introductory information about ASSIST by using -s.

When you invoke *assist*, you perform operations within the program by using *assist* commands. To see a list of the *assist* commands, press the Control-A key combination when you are in *assist*. When you do this, a list of the commands is printed on the terminal screen.

EXAMPLE

This example illustrates how to go directly to a particular command form. In this case, *mkdir* is the desired command form.

```
assist mkdir
```

SEE ALSO

astgen(1) in the Programmer's Reference Manual
ASSIST Software User's Guide

[This page left blank.]

NAME

at, batch - execute commands at a later time

SYNOPSIS

at time [date] [+ increment]
at -r job ...
at -l [job ...]

batch

DESCRIPTION

The *at* and *batch* commands read commands from standard input to be executed at a later time. The *at* command allows you to specify when the commands should be executed, while jobs queued with *batch* execute when the system load level permits. The *at* command may be used with the following options:

-r Removes jobs previously scheduled with *at*.

-l Reports all jobs scheduled for the invoking user.

Standard output and standard error output are mailed to the user unless they are redirected elsewhere. The shell environment variables, current directory, *umask*, and *ulimit* are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use *at* if their name appears in the file **/usr/lib/cron/at.allow**. If that file does not exist, the file **/usr/lib/cron/at.deny** is checked to determine if the user should be denied access to *at*. If neither file exists, only root is allowed to submit a job. If **at.deny** exists and is empty, global usage is permitted. If **at.allow** exists and is empty, no usage is permitted. If **at.allow** exists, **at.deny** is ignored. The **allow/deny** files consist of one user name per line. These files can only be modified by the superuser.

The *time* may be specified as one, two, or four digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning *hour:minute*. A suffix **am** or **pm** may be appended; otherwise, a 24-hour clock time is understood. The suffix **zulu** may be used to indicate GMT. The special names **noon, midnight, now,** and **next** are also recognized.

An optional *date* may be specified as either a month name followed by a day number (and possibly year number preceded by an optional comma) or a day of the week (fully spelled or abbreviated to three characters). Two special days, **today** and **tomorrow** are recognized. If no *date* is given, **today** is assumed if the given hour is greater than the current hour and **tomorrow** is assumed if it is less.

If the given month is less than the current month (and no year is given), next year is assumed.

The optional *increment* is simply a number suffixed by one of the following: **minutes, hours, days, weeks, months,** or **years.** (The singular form is also accepted.)

Thus legitimate commands include:

```
at 0815am Jan 24
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday
```

The *at* and *batch* commands write the job number and schedule time to standard error.

The *batch* command submits a batch job. It is almost equivalent to "at.now."

The **at -r** command removes jobs previously scheduled by *at* or *batch*. The job number is the number given to you previously by the *at* or *batch* command. You can also get job numbers by typing **at -l**. You can remove only your own jobs unless you are the superuser.

EXAMPLES

The *at* and *batch* commands read from standard input the commands to be executed at a later time. The *sh*(1) command provides a different way of specifying standard input. Within your commands, it may be useful to redirect standard output.

This sequence can be used at a terminal:

```
batch
sort filename >outfile
<control-D> (hold down control and depress D)
```

This sequence, which demonstrates redirecting standard error to a pipe, is useful in a shell procedure (the sequence of output redirection specifications is significant):

```
batch <<!
sort filename 2>&1 >outfile | mail loginid
!
```

To have a job reschedule itself, invoke *at* from within the shell procedure by including code similar to the following within the shell file:

echo "sh *shellfile*" | at 1900 thursday next week

FILES

/usr/lib/cron	main cron directory
/usr/lib/cron/at.allow	list of allowed users
/usr/lib/cron/at.deny	list of denied users
/usr/lib/cron/queue	scheduling information
/usr/spool/cron/atjobs	spool area

SEE ALSO

kill(1), mail(1), nice(1), ps(1), sh(1), sort(1)
cron(1M) in the Administrator's Reference Manual

DIAGNOSTICS

Diagnostics include various syntax errors and times out of range.

[This page left blank.]

NAME

awk - pattern scanning and processing language

SYNOPSIS

awk [**-Fc**] [prog] [parameters] [files]
awk [**-Fre**] [parameter...] ['prog'] [**-f** progfile] [file...]

DESCRIPTION

The *awk* language 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*. The *prog* string should be enclosed in single quotes (') to protect it from the shell.

The *parameters,* in the form x=... y=... etc., may be passed to *awk.*

Files are read in order; if there are no files, the standard input is read. The filename - 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; see the following text). 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 >*expr* 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*(3S) in the Programmer's Reference Manual].

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument is present. 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*(3S) 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* [see *grep*(1)]. 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 *relop* is any of the six relational operators in C, and *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 **-F***c* 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).

Internationalization

In addition to the ASCII character set, characters from supplementary code sets can be used in pattern-action statements and comments.

The field separators specified with option **-F** and the environment variables **OFS**, **ORS** and **Fs** can be characters from supplementary code sets.

In regular expressions, pattern searches are performed on characters, not bytes similarly to *ed*'s pattern processing.

The values returned by the built-in function, **length**(s), and the arguments *n* of the built-in function, substr(s, m, n), are the length of the EUC in bytes.

EXAMPLES

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 the previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page numbers starting at 5:

```
/Page/ { $2 = n++; }
       { print }
```

command line: awk -f program n=5 input

NOTE

The implementation of *awk* in MNLS 3.2 is based on *nawk* in System V Release 3.2.

SEE ALSO

grep(1), sed(1), nawk(1)
lex(1), printf(3S) in the Programmer's Reference Manual

BUGS

Input white space is not preserved on output if fields are involved.

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 the null string ("") to it.

NAME

banner - make posters

SYNOPSIS

banner strings

DESCRIPTION

The *banner* command prints its arguments (each up to 10 characters long) in large letters on the standard output. Spaces can be included in an argument by surrounding it with quotes. The maximum number of characters that can be accommodated in a line is implementation-dependent; excess characters are simply ignored.

Internationalization

This command has no international functionality.

SEE ALSO

echo(1)

[This page left blank.]

NAME

basename, dirname - deliver portions of pathnames

SYNOPSIS

basename string [suffix]
dirname string

DESCRIPTION

The *basename* command 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 (` `) within shell procedures.

The *dirname* command delivers all but the last level of the pathname in *string*.

EXAMPLES

The following example, invoked with the argument **/usr/src/cmd/cat.c**, compiles the named file and moves the output to a file named **cat** in the current directory:

```
cc $1
mv a.out `basename $1 '\.c' `
```

The following example sets the shell variable **NAME** to **/usr/src/cmd**:

```
NAME=`dirname /usr/src/cmd/cat.c`
```

SEE ALSO

sh(1)

[This page left blank.]

NAME

bc - arbitrary-precision arithmetic language

SYNOPSIS

bc [**-c**] [**-l**] [file ...]

DESCRIPTION

The *bc* command is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The *bc*(1) utility is actually a preprocessor for *dc*(1), which it invokes automatically unless the **-c** option is present. In this case the *dc* input is sent to the standard output instead. The options are as follows:

-c Compile only. The output is send to the standard output.

-l Argument stands for the name of an arbitrary precision math library.

The syntax for *bc* programs is as follows; the 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.

EXAMPLES

```
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.

FILES

/usr/lib/lib.b	mathematical library
/usr/bin/dc	desk calculator proper

SEE ALSO

dc(1)

BUGS

The *bc* command does not yet recognize the logical operators, && and ||.
The *for* statement must have all three expressions (E's).
The *quit* is interpreted when read, not when executed.

[This page left blank.]

NAME

bdiff - big diff

SYNOPSIS

bdiff file1 file2 [n] [-s]

DESCRIPTION

The *bdiff* command is used in a manner analogous to *diff*(1) to find which lines in two files must be changed to bring the files into agreement. Its purpose is to allow processing of files which are too large for *diff*.

The parameters to *bdiff* are:

file1 (file2)
: The name of a file to be used. If *file1 (file2)* is -, the standard input is read.

n
: The number of line segments. The value of *n* is 3500 by default. If the optional third argument is given and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail.

-s
: Specifies that no diagnostics are to be printed by *bdiff* (silent option). Note, however, that this does not suppress possible diagnostic messages from *diff*(1), which *bdiff* calls.

The *bdiff* command ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. If both optional arguments are specified, they must appear in the order indicated in the **SYNOPSIS**.

The output of *bdiff* is exactly that of *diff*, with line numbers adjusted to account for the segmenting of the files (that is, to make it look as if the files had been processed whole). Note that because of the segmenting of the files, *bdiff* does not necessarily find a smallest sufficient set of file differences.

FILES

/tmp/bd?????

SEE ALSO

diff(1)

[This page left blank.]

NAME

bfs - big file scanner

SYNOPSIS

bfs [-] name

DESCRIPTION

The *bfs* command is similar to *ed*(1) except that it is read only and processes much larger files. Files can be up to 1024K bytes and 32K lines, with up to 512 characters per line, including newline. The *bfs* command is usually more efficient than *ed*(1) for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the **w** command. The optional - suppresses printing of sizes. Input is prompted with *, if P and a carriage return are typed, as in *ed*(1). Prompting can be turned off again by inputting another P and carriage return. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under *ed*(1) are supported, with the exception of the range constructions (. . .). In addition, regular expressions may be surrounded with two symbols besides / and ? ; the > sign indicates downward search without wrap-around, and < indicates upward search without wrap-around. There is a slight difference in mark names; only the letters **a** through **z** may be used, and all 26 marks are remembered.

The **e**, **g**, **v**, **k**, **p**, **q**, **w**, =, ! and null commands operate as described under *ed*(1), except that the default command list for **g** and **v** is the null command, not **p**. Commands such as **---**, **+ + +-**, **+ + + =**, **-12**, and **+4p** are accepted. Note that **1,10p** and **1,10** will both print the first ten lines. The **f** command only prints the name of the file being scanned; there is no remembered filename. The **w** command is independent of output diversion, truncation, or crunching (see the **xo**, **xt**, and **xc** commands in the following text). The following additional commands are available:

xf*file* Further commands are taken from the named *file*. When an end-of-file is reached, an interrupt signal is received or an error occurs; reading resumes with the file containing the **xf**. The **xf** commands may be nested to a depth of 10.

xn List the marks currently in use (marks are set by the **k** command).

xo [*file*]

Further output from the **p** and null commands is diverted to the named *file*, which, if necessary, is created in mode 666 (readable and writable by everyone), unless your *umask* setting [see *umask*(1)] dictates otherwise. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

:*label*

This positions a *label* in a command file. The *label* is terminated by newline, and blanks between the **:** and the start of the *label* are ignored. This command may also be used to insert comments into a command file, since labels need not be referenced.

(.,.) **xb**/*regular expression*/*label*

A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

1. Either address is not between **1** and **$**.
2. The second address is less than the first.
3. The regular expression does not match at least one line in specified range, including the first and last lines.

If the command succeeds, **.** is set to the line matched and a jump is made to *label*. This command is the only one that does not issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. Note that the command

```
xb/^/ label
```

is an unconditional jump.

The **xb** command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe, only a downward jump is possible.

xt*number*

Output from the **p** and null commands is truncated to at most *number* characters. The initial number is 255.

xv [*digit*] [*spaces*] [*value*]

The variable name is the specified *digit* following the **xv**. The commands **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. The command **xv61,100p** assigns the value **1,100p** to the variable **6**.

To reference a variable, put a % in front of the variable name. For example, using the above assignments for variables 5 and 6:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would globally search for the characters 100 and print each line containing a match. To escape the special meaning of %, a \ must precede it.

```
g/".*\%[cds]/p
```

could be used to match and list lines containing *printf* of characters, decimal integers, or strings.

Another feature of the xv command is that the first line of output from a UNIX system command can be stored into a variable. The only requirement is that the first character of *value* be an !. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable 5, print it, and increment the variable 6 by one. To escape the special meaning of ! as the first character of *value*, precede it with a \.

```
xv7\!date
```

stores the value !date into variable 7.

xbz *label* and **xbn** *label*

These two commands will test the last saved *return code* from the execution of a UNIX system command (**!***command*) or non-zero value, respectively, to the specified label. The two examples below both search for the next five lines containing the string size.

```
xv55
: 1
```

```
/size/
xv5!expr %5 - 1
!if 0%5 != 0 exit 2
xbn 1

xv45
: 1
/size/
xv4!expr %4 - 1
!if 0%4 = 0 exit 2
xbz 1
```

xc [*switch*]
If *switch* is **1**, output from the **p** and null commands is crunched; if *switch* is **0**, it is not. Without an argument, **xc** reverses *switch*. Initially *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

Internationalization

The *bfs* command can process ASCII characters as well as characters from supplementary code sets in the text.

The *bfs* command can also recognize *labels* containing characters from supplementary code sets for **:**, **xb**, **xbn** and **xbz** commands.

Regular expression searches are performed on characters, not on individual bytes. Refer to *ed*(1).

The value designated by *number* with the **xt** command must be the number of displayed columns, not the number of characters or the number of EUC bytes.

Marks set by the **k** command must be ASCII characters in the range of *a* to *z*, and all 26 marks are remembered.

SEE ALSO

csplit(1), ed(1), umask(1) regcmp(3X) is the Programmer's Reference Manual

DIAGNOSTICS

The ? appears for errors in commands if prompting is turned off. Self-explanatory error messages appear when prompting is on.

WARNINGS

Size indication

The size of the file displayed at first and after read/write by the **e** or **w** commands is in bytes, not characters.

NAME

cal - print calendar

SYNOPSIS

cal [[month] year]

DESCRIPTION

The *cal* command prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. The *year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and the United States.

EXAMPLES

An unusual calendar is printed for September 1752. That is the month 11 days were skipped to make up for lack of leap year adjustments. To see this calendar, type: **cal 9 1752**

BUGS

The year is always considered to start in January, even though this is historically naive.

Beware that cal 88 refers to the early Christian era, not the 20th century.

[This page left blank.]

NAME

calendar - reminder service

SYNOPSIS

calendar [-]

DESCRIPTION

The *calendar* command 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 Aug. 24, august 24, 8/24, etc., are recognized, but not 24 August or 24/8. 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 or her login directory and sends them any positive results by *mail*(1). Normally this is done daily by facilities in the UNIX operating system.

FILES

/usr/lib/calprog	to figure out today's and tomorrow's dates
/etc/passwd	
/tmp/cal*	

SEE ALSO

mail(1)

BUGS

Your calendar must be public information for you to get reminder service. The *calendar*'s extended idea of tomorrow does not account for holidays.

[This page left blank.]

NAME

cat - concatenate and print files

SYNOPSIS

cat [**-u**] [**-s**] [**-v** [**-t**] [**-e**]] file ...

DESCRIPTION

The *cat* command reads each *file* in sequence and writes it on the standard output. Thus:

cat file

prints **file** on your terminal, and:

cat file1 file2 >file3

concatenates **file1** and **file2**, and writes the results in **file3**.

If no input file is given, or if the argument - is encountered, *cat* reads from the standard input file.

The following options apply to *cat*:

-u The output is not buffered. (The default is buffered output.)

-s The *cat* command is silent about nonexistent files.

-v Causes non-printing characters (with the exception of tabs, newlines and form-feeds) to be printed visibly. The ASCII control characters (octal 000 - 037) are printed as ^*n*, where *n* is the corresponding ASCII character in the range octal 100 - 137 (@, A, B, C, . . ., X, Y, Z, [, \,], ^, and _); the DEL character (octal 0177) is printed ^?. Other non-printable characters are printed as M-*x*, where *x* is the ASCII character specified by the low-order seven bits.

The following options may be used with the **-v** option:

-t Causes tabs to be printed as ^*I*'s and form-feeds to be printed as ^*L*'s.

-e Causes a $ character to be printed at the end of each line (prior to the newline).

The **-t** and **-e** options are ignored if the **-v** option is not specified.

Internationalization

The *cat* command can read and write characters from supplementary code sets.

SEE ALSO

cp(1), pg(1), pr(1)

WARNING

Redirecting the output of *cat* onto one of the files being read will cause the loss of the data originally in the file being read. For example, typing:

```
cat file1 file2 >file1
```

will cause the original data in **file1** to be lost.

When invoked with the **-v** option, *cat* considers all multibyte characters to be printable.

NAME

cd - change working directory

SYNOPSIS

cd [directory]

DESCRIPTION

If *directory* is not specified, the value of shell parameter **$HOME** is used as the new working directory. If *directory* specifies a complete path starting with /, *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory relative to one of the paths specified by the **$CDPATH** shell variable. The **$CDPATH** has the same syntax as, and similar semantics to, the **$PATH** shell variable. The *cd* command 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; therefore, it is recognized and is internal to the shell.

SEE ALSO

pwd(1), sh(1)
chdir(2) in the Programmer's Reference Manual

[This page left blank.]

NAME

chmod - change mode

SYNOPSIS

chmod mode file ...
chmod mode directory ...

DESCRIPTION

The permissions of the named *files* or *directories* are changed according to mode, which may be symbolic or absolute. Absolute changes to permissions are stated using octal numbers:

chmod *nnn file(s)*

where *n* is a number from 0 to 7. Symbolic changes are stated using mnemonic characters:

chmod *a operator b file(s)*

where *a* is one or more characters corresponding to **user**, **group**, or **other**; where *operator* is +, -, and =, signifying assignment of permissions; and where *b* is one or more characters corresponding to the type of permission.

An absolute mode is given as an octal number constructed from the "or" of the following modes:

4000	set user ID on execution
20#0	set group ID on execution if # is **7**, **5**, **3**, or **1** enable mandatory locking if # is **6**, **4**, **2**, or **0**
1000	sticky bit is turned on [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

Symbolic changes are stated using letters that correspond both to access classes and to the individual permissions themselves. Permissions to a file may vary depending on your user identification number (UID) or group identification number (GID). Permissions are described in three sequences each having three characters:

User	Group	Other
rwx	rwx	rwx

This example (meaning that User, Group, and Others all have reading, writing, and execution permission to a given file) demonstrates two categories for granting permissions, the access class and the permissions themselves.

Thus, to change the mode of a file's (or directory's) permissions using *chmod*'s symbolic method, use the following syntax for mode:

[who] operator [permission(s)], ...

A command line using the symbolic method would appear as follows:

chmod g+rw *file*

This command would make *file* readable and writable by the group.

The *who* part can be stated as one or more of the following letters:

u	user's permissions
g	group's permissions
o	others permissions

The letter **a** (all) is equivalent to **ugo** and is the default if *who* is omitted.

The *operator* can be + to add *permission* to the file's mode, - to take away *permission*, or = to assign *permission* absolutely. (Unlike other symbolic operations, = has an absolute effect in that it resets all other bits.) Omitting *permission* is only useful with = to take away all permissions.

The *permission* is any compatible combination of the following letters:

r	reading permission
w	writing permission
x	execution permission
s	user or group set-ID is turned on
t	sticky bit is turned on
l	mandatory locking will occur during access

Multiple symbolic modes separated by commas may be given, though no spaces may intervene between these modes. Operations are performed in the order given. Multiple symbolic letters following a single operator cause the corresponding operations to be performed simultaneously. The letter **s** is only meaningful with **u** or **g**, and **t** only works with **u**.

Mandatory file and record locking (**l**) refers to a file's ability to have its reading or writing permissions locked while a program is accessing that file. It is not possible to permit group execution and enable a file to be locked on execution at the same time. In addition, it is not possible to turn on the set-group-ID and

enable a file to be locked on execution at the same time. The following examples,

chmod g+x,+l *file*
chmod g+s,+l *file*

are, therefore, illegal usages and will elicit error messages.

Only the owner of a file or directory (or the superuser) may change a file's mode. Only the superuser may set the sticky bit on a non-directory file. In order to turn on a file's set-group-ID, your own group ID must correspond to the file's, and group execution must be set.

EXAMPLES

The first examples deny execution permissions to all. The absolute (octal) example permits only read permissions:

chmod a-x *file*
chmod 444 *file*

The next examples make a file readable and writable by the group and others:

chmod go+rw *file*
chmod 606 *file*

This example causes a file to be locked during access:

chmod +l *file*

The last two examples enable all to read, write, and execute the file; and they turn on the set-group-ID.

chmod =rwx,g+s *file*
chmod 2777 *file*

NOTES

In a Remote File Sharing environment, you may not have the permissions that the output of the ls -l command leads you to believe. For more information, see the Administration Guide.

SEE ALSO

ls(1)
chmod(2) in the Programmer's Reference Manual

NAME

chown, chgrp - change owner or group

SYNOPSIS

chown owner file ...
chown owner directory ...
chgrp group file ...
chgrp group directory ...

DESCRIPTION

The *chown* command changes the owner of the *files* or *directories* to *owner*. The owner may be either a decimal user ID or a login name found in the password file.

The *chgrp* command changes the group ID of the *files* or *directories* to *group*. The group may be either a decimal group ID or a group name found in the group file.

If either command is invoked by other than the superuser, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Only the owner of a file (or the superuser) may change the owner or group of that file.

FILES

/etc/passwd
/etc/group

NOTES

In a Remote File Sharing environment, you may not have the permissions that the output of the ls -l command leads you to believe. For more information see the Administration Guide.

SEE ALSO

chmod(1)
chown(2), group(4), passwd(4) in the Programmer's Reference Manual

[This page left blank.]

NAME

clear - clear terminal screen

SYNOPSIS

/bin/clear

DESCRIPTION

The *clear* command clears your screen if it is possible to do so. It looks in the environment for the terminal type and uses *terminfo* to figure out how to clear the screen.

[This page left blank.]

NAME

cmp - compare two files

SYNOPSIS

cmp [**-l**] [**-s**] [**-a***n*] [**-o**] file1 file2

DESCRIPTION

The two files are compared. (If *file1* 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.

-a*n* Start the comparison at byte offset *n*, where *n* is an octal number. (Note that the byte offset will be the same for both files.)

-o Ignore time and date stamp differences when comparing the contents of binary files.

SEE ALSO

comm(1), diff(1)

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

[This page left blank.]

NAME

col - filter reverse line-feeds

SYNOPSIS

col [-b] [-f] [-x] [-p]

DESCRIPTION

The *col* command reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse line feeds (ASCII code ESC-7), and by forward and reverse half-line-feeds (ESC-9 and ESC-8). The *col* command is particularly useful for filtering multicolumn output made with the **.rt** command of *nroff* and output resulting from use of the *tbl*(1) preprocessor.

If the **-b** option is given, *col* assumes that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read is output.

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.

Unless the **-x** option is given, *col* converts white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters SO (\017) and SI (\016) are assumed by *col* to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output, SI and SO characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, newline, SI, SO, VT (\013), and ESC followed by 7, 8, or 9. The VT character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

Normally, *col* ignores any escape sequences unknown to it that are found in its input; the **-p** option may be used to cause *col* to output these sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless the user is fully aware of the textual position of the escape sequences.

NOTES

The input format accepted by *col* matches the output produced by *nroff* with either the **-T37** or **-Tlp** options. Use **-T37** (and the **-f** option of *col*) if the ultimate disposition of the output of *col* is a device that can interpret half-line motions; use **-Tlp** otherwise.

BUGS

Cannot back up more than 128 lines.

Allows at most 800 characters, including backspaces, on a line.

Local vertical motions that result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NAME

comm - select or reject lines common to two sorted files

SYNOPSIS

comm [- [123]] file1 file2

DESCRIPTION

The *comm* command reads *file1* and *file2*, which should be ordered in ASCII collating sequence [see *sort*(1)], 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* prints nothing.

SEE ALSO

cmp(1), diff(1), sort(1), uniq(1)

[This page left blank.]

NAME

copy - copy groups of files

SYNOPSIS

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 as the source. In addition, the superuser may set the user and group ID. The owner and mode are not changed if the destination file exists. Note that there may be more than one source directory. If so, the effect is the same as if the *copy* command had been issued for each source directory with the same destination directory for each copy.

All options must be given as separate arguments, and they may appear in any order. The options are:

-a Asks the user before attempting a copy. If the response does not begin with a "y," then a copy is not done. This option also sets the **-ad** option.

-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 does not change the destination file. The **-n** flag is meaningless for directories. For special files, an **-n** flag is assumed (i.e., the destination of a special file must not exist).

-o If set, then every file copied has its owner and group set to those of the source. If not set, then the file's owner is the user who invoked the program.

-m If set, then every file copied has its modification time and access time set to that of the source. If not set, then the modification time is 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 are ignored.

-ad Asks the user whether the **-r** flag applies when a directory is discovered. If the answer does not begin with a y, then the directory is ignored.

-v If the verbose option is set, messages are 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 are the same as for the *cp* command.

dest The destination must be either a file or directory that is different from the source.

If *source* and *destination* are anything but directories, then *copy* acts just like a *cp* command. If both are directories, then *copy* copies each file into the destination directory according to the flags that have been set.

NOTES

Special device files can be copied. When they are copied, any data associated with the specified device is not copied.

SEE ALSO

chmod(1), cp(1)

NAME

cp, ln, mv - copy, link, or move files

SYNOPSIS

cp file1 [file2 ...] target
ln [**-f**] file1 [file2 ...] target
mv [**-f**] file1 [file2 ...] target

DESCRIPTION

The *file1* is copied (linked, moved) to *target*. Under no circumstance can *file1* and *target* be the same [take care when using *sh*(1) metacharacters]. If *target* is a directory, then one or more files are copied (linked, moved) to that directory. If *target* is a file, its contents are destroyed.

If *mv* or *ln* determines that the mode of *target* forbids writing, it will print the mode [see *chmod*(2)], ask for a response, and read the standard input for one line. If the line begins with **y**, the *mv* (move) or *ln* (link) occurs, if permissible; if not, the command exits. For *mv*, when source parent directories or the target directory is writable and has the sticky bit set, any of the following conditions must be true:

the user must own the file
the user must own the directory
the file must be writable to the user
the user must be the superuser

When the **-f** option is used or if the standard input is not a terminal, no questions are asked and the *mv* or *ln* is done.

Only *mv* allows *file1* to be a directory, in which case the directory rename occurs only if the two directories have the same parent; *file1* is renamed *target*. If *file1* is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file.

When using *cp*, if *target* is not a file, a new file is created which has the same mode as *file1* except that the sticky bit is not set unless you are superuser; the owner and group of *target* are those of the user. If *target* is a file, copying a file into *target* does not change its mode, owner, or group. The last modification time of *target* (and last access time, if *target* did not exist) and the last access time of *file1* are set to the time the copy was made. If *target* is a link to a file, all links remain and the file is changed.

SEE ALSO

chmod(1), cpio(1), rm(1)

WARNINGS

The *ln* command does not link across file systems. This restriction is necessary because file systems can be added and removed. When the destination of a copy is a file that already exists, **cp** will try to overwrite it; this preserves the destination file's ownership and so forth. If the destination file has an ownership you do not want, remove the destination file before doing the copy.

BUGS

If *file1* and *target* 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.

NAME

cpio - copy file archives in and out

SYNOPSIS

cpio -o [**acBQvV**] [**-T** blocksize in Kbytes] [**-C** bufsize] [[**-O** file] [**-M** message]]
cpio -i [**BcdmQrtuvVfsSb6k**] [**-T** blocksize in Kbytes] [**-C** bufsize] [[**-I** file] [**-M** message]] [pattern ...]
cpio -p [**adlmuvV**] directory

DESCRIPTION

The command *cpio -o* (copy out) reads the standard input to obtain a list of pathnames and copies those files onto the standard output together with pathname and status information. Output is padded to a 512-byte boundary by default.

The command *cpio -i* (copy in) extracts files from the standard input, which is assumed to be the product of a previous *cpio -o*. Only files with names that match *patterns* are selected. The *patterns* are regular expressions given in the filename-generating notation of *sh*(1). In *patterns*, metacharacters ?, *, and [...] match the slash (/) character, and backslash (\) is an escape character. A ! metacharacter means not. (For example, the !abc* pattern would exclude all files that begin with abc.)

Multiple *patterns* may be specified and if no *patterns* are specified, the default for *patterns* is * (i.e., select all files). Each *pattern* must be enclosed in double quotes; otherwise, the name of a file in the current directory is used. Extracted files are conditionally created and copied into the current directory tree based upon the options described in the following text. The permissions of the files will be those of the previous *cpio -o* . The owner and group of the files will be that of the current user unless the user is a superuser, which causes *cpio* to retain the owner and group of the files of the previous *cpio -o* .

NOTE: If *cpio -i* tries to create a file that already exists and the existing file is the same age or newer, *cpio* will output a warning message and not replace the file. (The -u option can be used to unconditionally overwrite the existing file.)

The command *cpio -p* (pass) reads the standard input to obtain a list of pathnames of files that are conditionally created and copied into the destination *directory* tree based upon the options described in the following paragraph. Archives of text files created by *cpio* are portable between implementations of UNIX System V.

The meanings of the available options are:

-a Reset *access* times of input files after they have been copied. Access times are not reset for linked files when *cpio -pla* is specified.

-b Reverse the order of the *bytes* within each word. Use only with the **-i** option.

-B Input/output is to be blocked 5120 bytes to the record. The default buffer size is 512 bytes when this and the **-C** options are not used. (**-B** does not apply to the *pass* option; **-B** is meaningful only with data directed to or from a character-special device, e.g., **/dev/rdsk/f0q15dt**.)

-c Write header information in ASCII *character* form for portability. Always use this option when origin and destination machines are different types.

-C *bufsize*
Input/output is to be blocked *bufsize* bytes to the record, where *bufsize* is replaced by a positive integer. The default buffer size is 512 bytes when this and **-B** options are not used. (The **-C** option does not apply to the *pass* option; **-C** is meaningful only with data directed to or from a character-special device, e.g., **/dev/rmt/c0s0**.)

-d The *directories* are to be created as needed.

-f Copy in all *files* except those in *patterns*. (See the paragraph on *cpio -i* for a description of *patterns*.)

-I *file*
Read the contents of *file* as input. If *file* is a character-special device, when the first medium is full, replace the medium and type a carriage return to continue to the next medium. Use only with the **-i** option.

-k Attempt to skip corrupted file headers and I/O errors that may be encountered. If you want to copy files from a medium that is corrupted or out of sequence, this option lets you read only those files with good headers. (For *cpio* archives that contain other *cpio* archives, if an error is encountered, *cpio* may terminate prematurely. The *cpio* command will find the next good header, which may be one for a smaller archive, and terminates when the smaller archive's trailer is encountered.) Used only with the **-i** option.

-l Whenever possible, *link* files rather than copy them. Usable only with the **-p** option.

-m Retain previous file *modification* time. This option is ineffective on directories that are being copied.

-M *message*
Define a message to use when switching media. When you use the **-O** or **-I** options and specify a character-special device, you can use this option to define the message that is printed when you reach the end of the medium.

One **%d** can be placed in the message to print the sequence number of the next medium needed to continue.

-O *file*
Direct the output of cpio to *file*. If *file* is a character-special device, when the first medium is full, replace the medium and type a carriage return to continue to the next medium. Use only with the **-o** option.

-Q Input/output is to be blocked 65,536 bytes to the record. Works like the **-B** option, with which it is mutually exclusive. The **-Q** option optimizes quarter-inch tape access.

-r Interactively *rename* files. If the user types a null line, the file is skipped. If the user types a ".", the original pathname will be copied. (Not available with *cpio -p*.)

-s *swap* bytes within each half word. Use only with the **-i** option.

-S *Swap* half words within each word. Use only with the **-i** option.

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

-T Input/output is blocked by an integer following T * 1024 bytes. Works like the **-B** option with which it is mutually exclusive and allows random block sizes in increments of 1024 bytes.

-u Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).

-v *verbose*; causes a list of filenames to be printed. When used with the **-t** option, the table of contents looks like the output of an **ls -l** command [see *ls*(1)].

-V *Special*Verbose: print a dot for each file seen. Useful to assure the user that *cpio* is working without printing out all filenames.

-6 Process an old (i.e., UNIX System *Sixth* Edition format) file. Use only with the **-i** option.

NOTE: The *cpio* command assumes 4-byte words.

If *cpio* reaches end of medium (end of a diskette for example) when writing to (**-o**) or reading from (**-i**) a character-special device, and **-O** and **-I** are not used, *cpio* will print the message:

```
If you want to go on, type device/filename when ready.
```

To continue, you must replace the medium and type the character-special device name (**/dev/rdsk/f0q15dt** for example) and a carriage return. You may want to continue by directing *cpio* to use a different device. For example, if you have two floppy drives, you may want to switch between them so *cpio* can proceed while you are changing the floppies. A carriage return alone causes the *cpio* process to exit.

Internationalization

The *cpio* command can process files containing characters from supplementary code sets. In pattern processing using metacharacters, matching is performed on characters, not bytes.

The *message* with the **-M** option can include characters from supplementary code sets.

EXAMPLES

The following examples show three uses of *cpio*.

When standard input is directed through a pipe to *cpio -o*, it groups the files so they can be directed (>) to a single file (*../newfile*). The **-c** option insures that the file will be portable to other machines. Instead of *ls*(1), you could use *find*(1), *echo*(1), *cat*(1), etc., to pipe a list of names to *cpio*. You could direct the output to a device instead of a file.

ls | cpio -oc > *../newfile*

The command *cpio -i* uses the output file of *cpio -o* (directed through a pipe with **cat** in the example), extracts those files that match the patterns (**memo/a1**, **memo/b***), creates directories below the current directory as needed (**-d** option), and places the files in the appropriate directories. The **-c** option is used when the file is created with a portable header. If no patterns were given, all files from *newfile* would be placed in the directory.

cat newfile | cpio -icd "*memo/a1*" "*memo/b**"

The command *cpio -p* takes the filenames piped to it and copies or links (**-l** option) those files to another directory on your machine (*newdir* in the example). The **-d** option says to create directories as needed. The **-m** option says retain the modification time. [It is important to use the **-depth** option of *find*(1) to generate pathnames for *cpio*. This eliminates problems *cpio* could have trying to create files under read only directories.]

find . -depth -print | cpio -pdlmv *newdir*

NOTES

1) Pathnames are restricted to 256 characters.
2) Only the superuser can copy special files.
3) Blocks are reported in 512-byte quantities.
4) If a file has 000 permissions, contains more than 0 characters of data, and the user is not root, the file will not be saved or restored.

SEE ALSO

cat(1), echo(1), find(1), ls(1), sh(1), tar(1)
cpio(4) in the Programmer's Reference Manual

[This page left blank.]

NAME

crontab - user crontab file

SYNOPSIS

crontab [file]
crontab -r
crontab -l

DESCRIPTION

The *crontab* command copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. The **-r** option removes a user's crontab from the crontab directory. The **crontab -l** lists the crontab file for the invoking user.

Users are permitted to use *crontab* if their names appear in the file **/usr/lib/cron/cron.allow.** If that file does not exist, the file **/usr/lib/cron/cron.deny** is checked to determine if the user should be denied access to *crontab.* If neither file exists, only root is allowed to submit a job. If **cron.allow** exists and is empty, no usage is permitted. If **cron.allow** exists, **cron.deny** is ignored. If **cron.allow** does not exist and **cron.deny** exists but is empty, global usage is permitted. The **allow/deny** files consist of one user name per line.

A crontab file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
day of the week (0-6 with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values) or a list of elements separated by commas. An element is either a number or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to. For example, 0 0 1,15 * 1 would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example, 0 0 * * 1 would run a command only on Mondays).

The sixth field of a line in a crontab file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is

translated to a newline 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.

The shell is invoked from your **$HOME** directory with an **arg0** of **sh.** Users who desire to have their *.profile* executed must explicitly do so in the crontab file. The *cron* command supplies a default environment for every shell, defining **HOME, LOGNAME, SHELL(=/bin/sh),** and **PATH(=:/bin:/usr/bin:/usr/local/bin:/usr/lbin).**

If you do not redirect the standard output and standard error of your commands, any generated output or errors are mailed to you.

FILES

/usr/lib/cron	main cron directory
/usr/spool/cron/crontabs	spool area
/usr/lib/cron/log	accounting information
/usr/lib/cron/cron.allow	list of allowed users
/usr/lib/cron/cron.deny	list of denied users

SEE ALSO

cron(1M), sh(1)

WARNINGS

If you inadvertently enter the *crontab* command with no argument(s), do not attempt to get out with a <Ctrl> d. This will cause all entries in your *crontab* file to be removed. Instead, exit with a <Del>.

NAME

crypt - encode/decode

SYNOPSIS

crypt [password]
crypt [**-k**]

DESCRIPTION

The *crypt* command reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no argument is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. If the -k option is used, *crypt* will use the key assigned to the environment variable CRYPTKEY. The *crypt* command encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

Files encrypted by *crypt* are compatible with those treated by the editors *ed*(1), *edit*(1), *ex*(1), and *vi*(1) in encryption mode.

The security of encrypted files depends on three factors:

1. the fundamental method must be hard to solve
2. direct search of the key space must be infeasible
3. sneak paths by which keys or clear text can become visible must be minimized.

The *crypt* command 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 lowercase letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

If 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. The choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), edit(1), ex(1), makekey(1), ps(1), stty(1), vi(1)

WARNINGS

This command is provided with the Crypt Utilities, which is only available in the United States. If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files are decrypted correctly.

NAME

cut - cut out selected fields of each line of a file

SYNOPSIS

cut -c*list* [file ...]
cut -f*list* [**-d** *char*] [**-s**] [file ...]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (**-c** option), or the length can vary from line to line and be marked with a field delimiter character like *tab* (**-f** option). The *cut* command can be used as a filter; if no files are given, the standard input is used. In addition, a filename of - explicitly refers to standard input.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional - to indicate ranges [e.g., **1,4,7**; **1-3,8**; **-5,10** (short for **1-5,10**); or **3-** (short for third through last field)].

-c*list* The *list* following **-c** (no space) specifies character positions (e.g., **-c1-72** would pass the first 72 characters of each line).

-f*list* The *list* following **-f** is a list of fields assumed to be separated in the file by a delimiter character (see **-d**); e.g., **-f1,7** copies the first and seventh field only. Lines with no field delimiters are passed through intact (useful for table subheadings), unless **-s** is specified.

-d*char* The character following **-d** is the field delimiter (**-f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

-s Suppresses lines with no delimiter characters in case of **-f** option. Unless specified, lines with no delimiters are passed through untouched.

Either the **-c** or **-f** option must be specified.

Use *grep*(1) to make horizontal cuts (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

NAME

csh - invoke a shell command interpreter that uses C-like syntax

SYNOPSIS

csh [**-cefinstvVxX**] [*arg* ...]

DESCRIPTION

The *csh* command language interpreter 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 then begins 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 user's home directory.

Lexical Structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters &, |, ;, <, >, (, and) form separate words. If doubled (for example, &&, ||, <<, or >>) these character pairs form single words. These parser metacharacters may be made part of other words, or you can take away 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 single or double quotations form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. The semantics of these quotations are described below. Within pairs of \ or " characters, a newline preceded by a \ gives a true newline character.

When the shell's input is not from the console, the character # introduces a comment which continues to the end of the input line. It does not have this special meaning when preceded by \ and placed inside the quotation marks `, ', 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 a semicolon (;) and are then executed sequentially. A sequence of pipeline commands may be executed without waiting for them 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 parentheses 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

The following sections 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 a \ 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 that contains a history substitution is echoed on the terminal before it is executed as it could have been typed without a history substitution.

Commands input from the terminal that 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 example, 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 a ! in the prompt string.

With the current event 13, its possible to 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 re-introduce the words of the specified events, each separated by a single blank. As a special case, !! refers to the previous command; so the !! command alone is essentially a *redo*. The form !# references the current command (the one being typed). 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 using a colon (:) and a designator for the desired words. The words of an 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 as follows:

Designator	**Description**
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 ^-$, or nothing if only one word in event
*x**	Abbreviates *x*-$
x-	Like *x** but omitting word $

The colon separating the event specification from the word designator can be omitted if the argument selector begins with a , $, *, - or %. After the optional word designator, a sequence of modifiers can be placed, each preceded by a :. The following modifiers are defined:

Modifier	**Description**
h	Removes a trailing pathname component
r	Removes a trailing .xxx component

s/*l*/*r*/	Substitutes *l* for *r*
t	Removes all leading pathname components
&	Repeats the previous substitution
g	Applies the change globally, prefixing the above
p	Prints the new command but does not execute it
q	Quotes the substituted words, preventing substitutions
x	Like q but breaks 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 sides 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 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 nonblank 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, ^lb^lib 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 that 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 's are prevented from any further interpretation. Strings enclosed in " are variable, and command expansion may occur.

In both cases, the resulting text becomes (all or part of) a single word; only in one special case (see Command Substitution 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 that 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". 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, you can alias print "'pr \!* | lpr'" to make a command that paginates its arguments to the lineprinter.

Variable Substitution

The shell maintains a set of variables, each of which has as its value 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 that causes command input to be echoed. The setting of this variable results from the -v command line option.

Other operations treat variables numerically. The at-sign (@) command permits numeric calculations to be performed and the result assigned to a variable. However, variable values are 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 dollar sign ($) characters. This expansion can be prevented by preceding the dollar sign with a backslash (\) except within double quotation marks (") where it *always* occurs, and within single quotation marks (') where it *never* occurs. Strings quoted by back quotation marks (`) are interpreted later (see "Command Substitution" in this manpage) so dollar sign substitution does not occur there until later, if at all. A dollar sign is passed unchanged if followed by a blank, tab, or end-of-line.

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

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

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

$*name*
${*name*}

Are replaced by the words of the value of variable *name,* each separated by a blank. Braces insulate *name* from following characters 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. However, modifiers and the other forms shown in the following list 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 variable 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. Only one : modifier is allowed on each $ expansion.

The following substitutions may not be modified with : modifiers.

$?name
${?name}
Substitutes the string 1 if name is set and 0 if it is not.

$?0 Substitutes 1 if the current input filename is known and 0 if it is not.

$$ Substitutes the (decimal) process number of the (parent) shell.

Command and Filename Substitution

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

Command Substitution

Command substitution is indicated by a command enclosed in back quotation marks. 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 replaces the original string. Within double quotation marks, 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 displays a complete line.

Filename Substitution

If a word contains any *, ?, [or { characters, 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 is replaced with an alphabetically sorted list of filenames that match the pattern. In a list of words specifying filename substitution, it is an error for no pattern to match an existing filename, but it is not required for each pattern to match. Only the metacharacters *, ?, and [imply pattern matching; the characters ~ and { are 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 it expands to the invoker's home directory as reflected in the value of the variable *home*. When followed by a name consisting of letters, digits, and - characters, the shell searches for a user with that name and substitutes the user's 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 does not appear at the beginning of a word, it is left unchanged.

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/s1/{oldls,ls}.c expands to /usr/source/s1/oldls.c /usr/source/s1/ls.c, whether or not these files exist, without any chance of error if the home directory for source is /usr/source. Similarly ../{memo,*box} might expand to ../memo ../box ../mbox. (Note that memo was not sorted with the results of matching *box.) As a special case {, }, and {} are passed unchanged.

Input/Output

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

< name

Opens file *name* (which is first variable, command and filename expanded) as the standard input.

< < word

Reads the shell input up to a line which is identical to *word*. The *word* is not subjected to variable, filename, or command substitution, and each input line is compared to *word* before any substitutions are made on this input line. Unless a quoting backslash, double or single quotation mark, or a back quotation mark appears in *word*, variable and command substitution is performed on the intervening lines, allowing \ to quote $, \ and . Commands that are substituted have all blanks, tabs, and newlines preserved except for the final newline, which is dropped. The resulting 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, it is truncated, and its previous contents are lost.

If the variable *noclobber* is set, then the file must not already exist or it must 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 to suppress this check.

The forms involving & route the diagnostic output into the specified file as well as the standard output. The *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, it is 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 built-in commands (to be described later) 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, with the following operators forming groups at the same level:

= = and ! =
< =, > =, <, and >
< < and > >
+ and -
* / and %

The = = and ! = operators compare their arguments as strings; all others operate on numbers. Strings that begin with 0 are considered octal numbers. Null or missing arguments are considered 0. The results 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; they should be surrounded by spaces except when adjacent to components of expressions which are syntactically significant to the parser [& ¦ < > ( )].

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 the following characters:

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, 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 that can be used to control command files (shell scripts) and, in limited but useful ways, 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 command line.

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 commands will succeed on nonseekable inputs.)

Built-In Commands

Built-in commands are executed within the shell. If a built-in command occurs as any component of a pipeline except the last, then it is executed in a subshell. The following list describes the syntax and function of the built-in commands:

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. The *name* is not allowed to be *alias* or *unalias*.

break

Causes execution to resume after the *end* of the nearest enclosing *foreach* or *while* statement. The remaining commands on the current line are executed. Multilevel 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*
Changes the shell's working directory to directory *name*. If no argument is given, then it changes 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
Continues 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 shell's standard output. A \c causes the echo to complete without printing a newline. A \n in *wordlist* causes a newline to be printed. Otherwise, the words are echoed, separated by spaces.

else
end
endif
endsw
See the following descriptions of the *foreach*, *if*, *switch*, and *while* statements.

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 built-in command *continue* may be used to continue the loop prematurely, and the built-in command *break* may be used 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.

glob *wordlist*

Like *echo* but no \ escapes are recognized and words are delimited by null characters in the output. Useful for programs that want 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.

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*.)

logout

Terminates a login shell. The only way to log out 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 superuser may specify negative niceness by using "nice -number ...." The command is always executed in a subshell, and the restrictions placed 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. Unless the shell is running detached, *nohup* has no effect. All processes detached with & are automatically run with *nohup*. (Thus *nohup* is not really needed.)

onintr
onintr -
onintr *label*

Controls the action of the shell on interrupts. The first form restores the default action of the shell on interrupts: 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 directories' contents 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 be necessary only 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 redirection 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 that have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index* 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*

Sets the value of the 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* that 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 match the *unalias* pattern.

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 undesirable 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 nonzero, the commands between the *while* and the matching end are evaluated. Use *break* and *continue* 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 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* argument of *name*. Both *name* and its *index* component must already exist.

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

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

Predefined 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 done explicitly by the user.

Variable	Description
argv	Set to the arguments of the shell; from this variable, positional parameters are substituted, i.e., $1 is replaced by $*argv*[1].
cdpath	Gives a list of alternate directories searched to find subdirectories in *cd* 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 nonbuilt-in commands, all expansions occur before echoing. Built-in 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 that has been referenced in this many events will not be discarded. A *history* that is too large may run the shell out of memory. The last executed command is always saved on the history list.
home	The home directory of the user, initialized from the environment. The filename expansion of ~ refers to this variable.
ignoreeof	If set, the shell ignores the end-of-file from input devices that are terminals. This prevents a shell from accidentally being terminated by typing CTRL-D.
mail	The files where the shell checks for mail. This is done after each command completion results in a prompt, if a specified interval has elapsed. The shell sends the message "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 sends the message "New mail in *name*" when there is mail in the file *name*.

noclobber 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 that are not dealing with filenames or after a list of filenames has been obtained and further expansions are not desirable.

nomatch 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 pathnames will execute. The usual search path is **/bin**, **/usr/bin**, and ., but this may vary from system to system. For the superuser, the default search path is **/etc**, **/bin** and **/usr/bin**. A shell that is given neither the **-c** nor the **-t** option will normally hash the contents of the directories in the *path* variable after reading **.cshrc** and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash*, or the commands may not be found.

prompt The string that is printed before each command is read from an interactive terminal input. If a ! appears in the string it will be replaced by the current event number unless a preceding \ is given. The default is % or # for the superuser.

shell The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set but are not executable by the system. (See the section "Nonbuilt-In Command Execution" in this manpage.) The *shell* is 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. Abnormal

termination results in a core dump. Built-in commands that fail return exit status 1; all other built-in commands set status 0.

time Controls automatic timing of commands. If set, then any command that takes more than this many CPU seconds will cause a line giving user, system, and real times and a utilization percentage (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.

The shell copies the environment variable PATH into the variable *path* and copies the value back into the environment whenever *path* is set. Thus, it is not necessary to worry about its setting other than in the file **.cshrc** as inferior *csh* processes will import the definition of *path* from the environment.

Nonbuilt-In Command Execution

When a command to be executed is found to not be a built-in command, the shell attempts to execute the command via *exec*(S). 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 pathname 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 *cd* 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 and 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 pathname 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:

Flag	Description
-c	Reads commands from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.
-e	Causes the shell to exit if any invoked command terminates abnormally or yields a nonzero exit status.
-f	Lets the shell start faster because it will neither search for nor execute commands from the file **.cshrc** in the user's home directory.
-i	Makes the shell interactive. The shell prompts for its top-level input even if it appears not to be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.
-n	Causes commands to be parsed but not executed. This may aid in syntactic checking of shell scripts.
-s	Causes command input to be taken from the standard input.
-t	Reads and executes a single line of input. A backslash (\) can 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	Causes the *echo* variable to be set even before **.cshrc** is executed.

After processing of flag arguments, if arguments remain but none of the **-c**, **-i**, **-s**, or **-t** options were 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 on a typical system most shell scripts are written for the standard shell [see *sh*(1)], the C shell executes such a standard shell if the first character of a script is not a #: that is, 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 that the shell inherited from its parent. The shell's 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**.

NEW ENVIRONMENT VARIABLES

The new environment variable described in this section has been added to the C shell. The C shell will behave normally for those users who do not set **DOSPATH**. Users who want to execute DOS programs directly from the C shell, that is, bypassing the normal DOS bootup that occurs when running *vpix*, should set **DOSPATH** to include those directories in **PATH** that contain DOS executables.

DOSPATH is a string with the same format as **PATH**; it contains a subset of the list of directories from **PATH**. When searching a directory in **PATH** for a program, the C shell determines whether that directory is also in **DOSPATH**. If it is not, the C shell acts as usual. If it is, the C shell looks first for the command with the suffix **.com**, then **.exe**, then **.bat**, and finally, for the command without any suffix. Whenever the result of a path search gives a file with one of these DOS suffixes, the shell runs the *vpix* program via a standard search path and adds arguments -c and the full pathname of the DOS program (including the suffix).

For example, if **PATH** is set to **:/bin:/usr/bin**, **DOSPATH** is set to ., the current directory is **/usr/john/dosbin**, and there is a DOS program named *abc.comi* in the current directory, then typing *abc* to the C shell will cause the command *vpix -c /usr/john/dosbin/abc.com* to be executed, which will run the DOS program *abc.com* without the normal *vpix* DOS bootup.

FILES

~/.cshrc	Read by each shell at the beginning of execution
~/.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
/etc/cshrc	Default file of automatically invoked commands

NOTES

Words can be no longer than 512 characters. The number of arguments to a command which involves filename expansion is limited to 1/6 number of characters allowed in an argument list, which is 5120 less the characters in the environment. 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* substitutions on a single line to 20.

Built-in control structure commands like *foreach* and *while* cannot be used with the pipe symbol (¦), ampersand (&), or semicolon (;).

Commands within loops prompted for by ? are not placed in the *history* list.

It is not possible to use the colon (:) modifiers on the output of command substitutions.

The *csh* interpreter attempts to import and export the PATH variable for use with regular shell scripts. This only works for simple cases, where the PATH contains no command characters.

This version of *csh* does not support or use the process control features of the 4th Berkeley Distribution.

You can modify the list of commands that *csh* automatically invokes by editing the **/etc/default/.cshrc** file. For example, if you want to automatically assign the **alias** *h* to the **history** command, add the following line to the **/etc/default/.cshrc** file using the computer editor of your choice:

```
alias history h
```

SEE ALSO

umask(1), wait(1)
access(2), exec(2), fork(2), pipe(2), ssignal(3C), a.out(4), environ(5) in the Programmer's Reference Manual

[This page left blank.]

NAME

csplit - context split

SYNOPSIS

csplit [-s] [-k] [-f prefix] file arg1 [... argn]

DESCRIPTION

The *csplit* command reads *file* and separates it into n+1 sections, defined by the arguments *arg1*... *argn*. By default the sections are placed in xx00 ... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.
:
:

n+1: From the line referenced by *argn* to the end of *file*.

If the *file* argument is a -, then standard input is used.

The options to *csplit* are:

-s The *csplit* command normally prints the character counts for each file created. If the -s option is present, *csplit* suppresses the printing of all character counts.

-k The *csplit* command normally removes created files if an error occurs. If the -k option is present, *csplit* leaves previously created files intact.

-f *prefix* If the **-f** option is used, the created files are named *prefix*00 ... *prefixn*. The default is xx00 ... xx*n*.

The arguments (*arg1* ... *argn*) to *csplit* can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or - some number of lines (e.g., **/Page/-5**).

%rexp% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded newlines. The *csplit* command does not affect the original file; it is the user's responsibility to remove it.

EXAMPLES

```
csplit -f cobol file '/procedure division/' /par5./ /par16./
```

The preceding example creates four files, **cobol00** ... **cobol03**. After editing the split files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that the preceding example overwrites the original file.

The next example would split the file at every 100 lines, up to 10,000 lines. The **-k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed:

```
csplit -k file 100 {99}
```

Assuming that **prog.c** follows the normal C coding convention of ending routines with a } at the beginning of the line, the following example creates a file containing each separate C routine (up to 21) in **prog.c**:

```
csplit -k prog.c '%main(%' '/^}/+1' {20}
```

SEE ALSO

ed(1), sh(1)
regexp(5) in the Programmer's Reference Manual

DIAGNOSTICS

Self-explanatory except for:

arg - out of range

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

ct - spawn getty to a remote terminal

SYNOPSIS

ct [**-wn**] [**-xn**] [**-h**] [**-v**] [**-s**speed] telno ...

DESCRIPTION

The *ct* command dials the telephone number of a modem that is attached to a terminal, and spawns a *getty* process to that terminal. The *telno* command is a telephone number, with equal signs for secondary dial tones and minus signs for delays at appropriate places. (The set of legal characters for *telno* is 0 through 9, -, =, *, and #. The maximum length *telno* is 31 characters). If more than one telephone number is specified, the *ct* command tries each in succession until one answers; this is useful for specifying alternate dialing paths.

The *ct* command will try each line listed in the file **/usr/lib/uucp/Devices** until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, *ct* asks if it should wait for one, and if so, for how many minutes it should wait before it gives up. The *ct* command continues to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the **-w***n* option, where *n* is the maximum number of minutes that *ct* is to wait for a line.

The **-x***n* option is used for debugging; it produces a detailed output of the program execution on stderr (standard error). The debugging level, *n*, is a single digit; **-x9** is the most useful value.

Normally, *ct* hangs up the current line, so the line can answer the incoming call. The **-h** option prevents this action. The **-h** option also waits for the termination of the specified *ct* process before returning control to the user's terminal. If the **-v** option is used, *ct* sends a running narrative to the standard error output stream.

The data rate may be set with the **-s** option, where *speed* is expressed in baud. The default rate is 1200.

After the user on the destination terminal logs out, there are two things that could occur depending on what type of getty is on the line (*getty* or *uugetty*). For the first case, *ct* prompts, `Reconnect?` If the response begins with the letter **n**, the line is dropped; otherwise, *getty* is started again and the **login:** prompt is printed. In the second case, there is already a getty (*uugetty*) on the line, so the **login:** message appears.

To log out properly, the user must type <Ctrl> d.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

FILES

/usr/lib/uucp/Devices
/usr/adm/ctlog

SEE ALSO

cu(1C), login(1), uucp(1C)
getty(1M), uugetty(1M) in the Administrator's Reference Manual

BUGS

For a shared port, one used for both dial-in and dial-out, the *uugetty* program running on the line must have the **-r** option specified [see *uugetty*(1M)].

NAME

cu - call another UNIX system

SYNOPSIS

cu [-sspeed] [-lline] [-h] [-t] [-d] [-o ¦ -e] [-n] telno
cu [-s speed] [-h] [-d] [-o ¦ -e] -l line
cu [-h] [-d] [-o ¦ -e] systemname

DESCRIPTION

The *cu* command calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files.

The *cu* command accepts the following options and arguments:

-s*speed* Specifies the transmission speed (300, 1200, 2400, 4800, 9600); the default value is "Any" speed which will depend on the order of the lines in the **/usr/lib/uucp/Devices** file. Most modems are either 300 or 1200 baud. Directly connected lines may be set to a speed higher than 1200 baud.

-l*line* Specifies a device name to use as the communication line. This can be used to override the search that would otherwise take place for the first available line having the right speed. When the **-l** option is used without the **-s** option, the speed of a line is taken from the **Devices** file. When the **-l** and **-s** options are both used together, *cu* will search the **Devices** file to check if the requested speed for the requested line is available. If so, the connection is made at the requested speed; otherwise, an error message is printed and the call is not made. The specified device is generally a directly connected asynchronous line (e.g., **/dev/tty***ab*) in which case a telephone number (*telno*) is not required. The specified device need not be in the **/dev** directory. If the specified device is associated with an auto dialer, a telephone number must be provided. Use of this option with *systemname* rather than *telno* does not give the desired result (see *systemname* in following text).

-h Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

-t Used to dial an ASCII terminal which has been set to auto answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.

-d Causes diagnostic traces to be printed.

-o Designates that odd parity is to be generated for data sent to the remote system.

-e Designates that even parity is to be generated for data sent to the remote system.

-n For added security, it prompts the user to provide the telephone number to be dialed rather than taking it from the command line.

telno When using an automatic dialer, the argument is the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of four seconds.

systemname A *uucp* system name may be used rather than a telephone number; in this case, *cu* will obtain an appropriate direct line or telephone number from **/usr/lib/uucp/Systems.** Note: The *systemname* option should not be used in conjunction with the **-l** and **-s** options as *cu* will connect to the first available line for the system name specified, ignoring the requested line and speed.

After making the connection, *cu* runs as two processes:

The *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system.

The *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output.

Normally, an automatic DC3/DC1 protocol is used to control input from the remote system so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following user-initiated commands:

~. terminate the conversation.

~! escape to an interactive shell on the local system.

~!*cmd*... run *cmd* on the local system (by **sh -c**).

~$*cmd*... run *cmd* locally and send its output to the remote system.

~%**cd** change the directory on the local system. Note: The ~!**cd** will cause the command to be run by a sub-shell, probably not what was intended.

~ %take *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~ %put *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places. For both ~ %take and put commands, as each block of the file is transferred, consecutive single digits are printed to the terminal.
~ ~ *line*	send the line ~ *line* to the remote system.
~ %break	transmit a BREAK to the remote system (which can also be specified as ~ %b).
~ %debug	toggles the -d debugging option on or off (which can also be specified as ~ %d).
~ t	prints the values of the termio structure variables for the user's terminal (useful for debugging).
~ l	prints the values of the termio structure variables for the remote communication line (useful for debugging).
~ %nostop	toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output. Internally the program accomplishes this by initiating an output diversion to a file when a line from the remote begins with ~ . The complete sequence is:

~ > [>]: *file*
zero or more lines to be written to *file*
~ >

Data from the remote is diverted (or appended, if > > is used) to *file* on the local system. The trailing ~ > marks the end of the diversion.

The use of ~ %put requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current control characters on the local system. Backslashes are inserted at appropriate places.

The use of ~ %take requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, tabs mode [see *stty*(1)] should be set on the remote system if tabs are to be copied without expansion to spaces.

When *cu* is used on system X to connect to system Y and subsequently used on system Y to connect to system Z, commands on system Y can be executed by using ~ ~. Executing a tilde command reminds the user of the local system uname. For example, *uname* can be executed on Z, X, and Y as follows:

```
uname
Z
~[X]!uname
X
~~[Y]!uname
Y
```

In general, ~ causes the command to be executed on the original machine, ~ ~ causes the command to be executed on the next machine in the chain.

EXAMPLES

To dial a system whose telephone number is 9 201 555 1212 using 1200 baud (where a dial tone is expected after the 9):

```
cu -s1200 9=12015551212
```

If the speed is not specified, "Any" is the default value.

To log in to a system connected by a direct line, enter:

```
cu -l /dev/ttyXX
```

or

```
cu -l ttyXX
```

To dial a system with the specific line and a specific speed, enter:

```
cu -s1200 -l ttyXX
```

To dial a system using a specific line associated with an auto dialer, enter:

```
cu -l culXX 9=12015551212
```

To use a system name, enter:

```
cu systemname
```

FILES

/usr/lib/uucp/Systems
/usr/lib/uucp/Devices
/usr/spool/locks/LCK..(tty-device)

SEE ALSO

cat(1), ct(1C), echo(1), stty(1), uucp(1C), uname(1)

DIAGNOSTICS

Exit code is zero for normal exit, otherwise, one.

WARNINGS

The *cu* command buffers input data internally and does not do any integrity checking on data it transfers. Data fields with special *cu* characters may not be transmitted properly. Depending on the interconnection hardware, it may be necessary to use a ~. to terminate the conversion even if **stty 0** has been used. Non-printing characters are not dependably transmitted using either the **~ %put** or **~ %take** commands. The *cu* command between some modems will not return a login prompt immediately upon connection. A carriage return will return the prompt.

BUGS

During the **~ %put** operation, there is an artificial slowing of transmission by *cu* so that loss of data is unlikely.

[This page left blank.]

NAME

cut - cut out selected fields of each line of a file

SYNOPSIS

cut -c*list* [file ...]
cut -f*list* [**-d** *char*] [**-s**] [file ...]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (**-c** option), or the length can vary from line to line and be marked with a field delimiter character like *tab* (**-f** option). The *cut* command can be used as a filter; if no files are given, the standard input is used. In addition, a filename of **-** explicitly refers to standard input.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional **-** to indicate ranges [e.g., **1,4,7**; **1-3,8**; **-5,10** (short for **1-5,10**); or **3-** (short for third through last field)].

-c*list* The *list* following **-c** (no space) specifies character positions (e.g., **-c1-72** would pass the first 72 characters of each line).

-f*list* The *list* following **-f** is a list of fields assumed to be separated in the file by a delimiter character (see **-d**); e.g., **-f1,7** copies the first and seventh field only. Lines with no field delimiters are passed through intact (useful for table subheadings), unless **-s** is specified.

-d*char* The character following **-d** is the field delimiter (**-f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

-s Suppresses lines with no delimiter characters in case of **-f** option. Unless specified, lines with no delimiters are passed through untouched.

Either the **-c** or **-f** option must be specified.

Use *grep*(1) to make horizontal cuts (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut -d: -f1,5 /etc/passwd	mapping of user IDs to names
name= `who am i | cut -f1 -d" "`	to set **name** to current login name.

SEE ALSO

grep(1), paste(1)

DIAGNOSTICS

ERROR: line too long
A line can have no more than 1023 characters or fields, or there is no newline character.

ERROR: bad list for c/f option
Missing **-c** or **-f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

ERROR: no fields The *list* is empty.

ERROR: no delimiter
Missing *char* on **-d** option.

ERROR: cannot handle multiple adjacent backspaces
Adjacent backspaces cannot be processed correctly.

WARNING: cannot open <filename>
Either *filename* cannot be read or does not exist. If multiple filenames are present, processing continues.

NAME

cw, checkcw – prepare constant-width text for troff

SYNOPSIS

cw [**-lxx**] [**-rxx**] [**-fn**] [**-t**] [**+t**] [**-d**] [files]

checkcw [**-lxx**] [**-rxx**] files

DESCRIPTION

The *cw* utility is a preprocessor for *troff* input files that contain text to be typeset in the constant-width (CW) font. Note that the *cw* utility has been superseded and is no longer required in the current version of Documenter's Workbench (2.0).

Text typeset with the CW font resembles the output of terminals and of line printers. This font is used to typeset examples of programs and computer output in user manuals, programming texts, and so forth. It has been designed to be quite distinctive (but not overly obtrusive) when used together with the Times Roman font.

Because the CW font contains a "non-standard" set of characters and because text typeset with it requires different character and inter-word spacing than is used for "standard" fonts, documents that use the CW font must be preprocessed by *cw*.

The CW font contains the 94 printing ASCII characters:

```
a b c d e f g h i j k l mn o p q r s t u v w x y z
ABCDEF GHI J KLMNOP QRS TUVWXYZ
0 1 2 3 4 5 6 7 8 9
! $ %&( ) ' ' * + @. , / : ; = ? [ ] | - _ ^ ~ " < > { } # \
```

plus eight non-ASCII characters represented by four-character *troff* names (in some cases attaching these names to "non-standard" graphics):

Character	Symbol	Troff Name
"Cents" sign	¢	\(ct
EBCDIC "not" sign	¬	\(no
Left arrow	←	\(<-
Right arrow	→	\(->
Down arrow	↓	\(da
Vertical single quote	'	\(fm
Control-shift indicator	†	\(dg
Visible space indicator	□	\(sq
Hyphen	-	\(hy

The hyphen is a synonym for the unadorned minus sign (-). Certain versions of *cw* recognize two additional names: \(ua for an up arrow (↑) and \(lh for a diagonal left-up (home) arrow.

The *cw* utility recognizes five request lines, as well as user-defined delimiters. The request lines look like *troff* macro requests, and are copied in their entirety by *cw* onto its output; thus, they can be defined by the user as *troff* macros; in fact, the .CW and .CN macros *should* be so defined (see "HINTS" in this manpage). The five requests are:

.CW Start of text to be set in the CW font; **.CW** causes a break; it can take precisely the same options, in precisely the same format, as are available on the *cw* command line.

.CN End of text to be set in the CW font; **.CN** causes a break; it can take the same options as are available on the *cw* command line.

.CD Change delimiters and/or settings of other options; takes the same options as are available on the *cw* command line.

.CP *arg1 arg2 arg3 ... argn*
All the arguments (which are delimited like *troff* macro arguments) are concatenated, with the odd-numbered arguments set in the CW font and the even-numbered ones in the prevailing font.

.PC *arg1 arg2 arg3 ... argn*
Same as **.CP**, except that the even-numbered arguments are set in the CW font and the odd-numbered ones in the prevailing font.

The .CW and .CN requests are meant to bracket text (for example, a program fragment) that is to be typeset in the CW font "as is." Normally, *cw* operates in the *transparent* mode. In that mode, except for the .CD request and the nine special four-character names listed in the table above, every character between .CW and .CN request lines stands for itself. In particular, *cw* arranges for periods (.) and apostrophes (') at the beginning of lines, and backslashes (\) everywhere to be "hidden" from *troff*. The transparent mode can be turned off (see below), in which case normal *troff* rules apply; in particular, lines that begin with . and ' are passed through untouched (except if they contain delimiters–see below). In either case, *cw* hides the effect of the font changes generated by the .CW and .CN requests; *cw* also defeats all ligatures (fi, ff, and so forth) in the CW font.

The only purpose of the .CD request is to allow the changing of various options other than just at the beginning of a document.

The user can also define *delimiters*. The left and right delimiters perform the same function as the .CW/.CN requests; they are meant, however, to enclose CW "words" or "phrases" in running text (see example under *BUGS* below). The *cw* preprocessor treats text between delimiters in the same manner as text enclosed by .CW/.CN pairs, except that, for aesthetic reasons, spaces and backspaces inside .CW/.CN pairs have the same width as other CW characters, while spaces and backspaces between delimiters are half as wide, so they have the same width as spaces in the prevailing text (but are *not* adjustable). Font changes due to delimiters are *not* hidden.

Delimiters have no special meaning inside .CW/.CN pairs.

The options are:

–l*xx* The one- or two-character string *xx* becomes the left delimiter; if *xx* is omitted, the left delimiter becomes undefined.

–r*xx* Same for the right delimiter. The left and right delimiters may (but need not) be different.

–f*n* The CW font is mounted in font position *n*; acceptable values for *n* are 1, 2, and 3 (default is 3, replacing the bold font). This option is only useful at the beginning of a document.

–t Turn transparent mode *off*.

+t Turn transparent mode *on* (this is the initial default).

–d Print current option settings on file descriptor 2 in the form of *troff* comment lines. This option is meant for debugging.

The *cw* preprocessor reads the standard input when no *files* are specified (or when – is specified as the last argument), so it can be used as a filter. Typical usage is:

cw *files* | troff ...

The *checkcw* utility checks that left and right delimiters, as well as the .CW/.CN pairs, are properly balanced. It prints out all offending lines.

HINTS

Typical definitions of the .CW and .CN macros meant to be used with the *mm*(5) macro package:

```
.de CW
.DS I
.ps 9
.vs 10.5p
.ta 16m/3u 32m/3u 48m/3u 64m/3u 80m/3u 96m/3u ...
..
.de CN
.ta .5i 1i 1.5i 2i 2.5i 3i ...
.vs
.ps
.DE
..
```

At the very least, the .CW macro should invoke the *troff* no-fill (**.nf**) mode.

When set in running text, the CW font is meant to be set in the same point size as the rest of the text. In displayed matter, on the other hand, it can often be profitably set one point *smaller* than the prevailing point size (the displayed definitions of .CW and .CN above are one point smaller than the running text on this page). The CW font is sized so that, when it is set in 9-point, there are 12 characters per inch.

Documents that contain CW text may also contain tables and/or equations. If this is the case, the order of preprocessing should be: *cw*, *tbl*, and *eqn*. Usually, the tables contained in such documents will not contain any CW text, although it is entirely possible to have *elements* of the table set in the CW font; of course, care must be taken that *tbl*(1) format information not be modified by *cw*. Attempts to set equations in the CW font are not likely to be either pleasing or successful.

In the CW font, overstriking is most easily accomplished with backspaces: letting the <left arrow> key represent a backspace, striking d, then the <left arrow> key twice, and then the <Ctrl>-<Shift> combination yields an overstrike over d. [Because backspaces are half as wide between delimiters as inside .CW/.CN pairs (see previous text), two backspaces are required for each overstrike between delimiters.]

FILES

/usr/lib/font/ftCW CW font-width table

SEE ALSO

Documenter's Workbench User's Guide
Documenter's Workbench Technical Discussion and Reference

WARNINGS

If text preprocessed by *cw* is to make any sense, it must be set on a typesetter equipped with the CW font or on a STARE facility; on the latter, the CW font appears as bold, but with the proper CW spacing.

BUGS

It is not a good idea to use periods (.), backslashes (\), or double quotes (") as delimiters, or as arguments to .CP and .PC.

Certain CW characters do not concatenate gracefully with certain Times Roman characters, for example, a CW ampersand (&) followed by a Times Roman comma (,); in such cases, judicious use of *troff* half- and quarter-spaces (\| and \^) is most salutary; for example, one should use _&_\^, (rather than just plain _&_,) to obtain &, (assuming that _ is used for both delimiters).

The *cw* utility is not compatible with *nroff*.

The output of *cw* is hard to read.

[This page left blank.]

NAME

date - print and set the date

SYNOPSIS

date [+ format]
date [mmddhhmm[[yy] | [ccyy]]]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set (only by the superuser). 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; *cc* is the century minus one and is optional; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. The *date* command takes care of the conversion to and from local standard and daylight saving time. Only the superuser may change the date.

If the argument begins with +, the output of *date* is under the control of the user. All output fields are of fixed size (zero-padded, if necessary). Each field descriptor is preceded by % and is replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a newline character. If the argument contains embedded blanks, it must be quoted (see "EXAMPLES" in this manpage).

Specifications of native language translations of month and weekday names are supported. The language used depends on the value of the environment variable **LANGUAGE** [see *environ*(5)]. The month and weekday names used for a language are taken from strings in the file for that language in the **/lib/cftime** directory [see *cftime*(4)].

After successfully setting the date and time, *date* displays the new date according to the format defined in the environment variable **CFTIME** [see *environ*(5)].

Field descriptors (must be preceded by a %):

- **a** abbreviated weekday name
- **A** full weekday name
- **b** abbreviated month name

B	full month name
d	day of month - 01 to 31
D	date as mm/dd/yy
e	day of month - 1 to 31 (single digits are preceded by a blank)
h	abbreviated month name (alias for %b)
H	hour - 00 to 23
I	hour - 01 to 12
j	day of year - 001 to 366
m	month of year - 01 to 12
M	minute - 00 to 59
n	insert a newline character
p	string containing ante-meridiem or post-meridiem indicator (by default, AM or PM)
r	time as *hh:mm:ss pp* where *pp* is the ante-meridiem or post-meridiem indicator (by default, AM or PM)
R	time as hh:mm
S	second - 00 to 59
t	insert a tab character
T	time as *hh:mm:ss*
U	week number of year (Sunday as the first day of the week) - 01 to 52
w	day of week - Sunday = 0
W	week number of year (Monday as the first day of the week) - 01 to 52
x	Country-specific date format
X	Country-specific time format
y	year within century - 00 to 99
Y	year as *ccyy* (4 digits)
Z	timezone name

EXAMPLE

date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'

would have generated as output:

DATE: 08/01/76
TIME: 14:45:05

FILES

/dev/kmem

NOTE

Administrators should note the following; if you attempt to set the current date to one of the dates that the standard and alternate time zones change (for example, the date that daylight time is starting or ending), and you attempt to set

the time to a time in the interval between the end of standard time and the beginning of the alternate time (or the end of the alternate time and the beginning of standard time), the results are unpredictable.

SEE ALSO

cftime(4), environ(5) in the Programmer's Reference Manual

DIAGNOSTICS

No permission	if you are not the superuser and you try to change the date
bad conversion	if the date set is syntactically incorrect
bad format character	if the field descriptor is not recognizable

[This page left blank.]

NAME

dc - desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

The *dc* command 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. [See *bc*(1), a preprocessor for *dc* that provides infix notation and a C-like syntax that implements functions. The *bc* command also provides reasonable control structures for programs.] 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 are printed.

q
: Exits the program. If executing a string, the recursion level is popped by two.

Q Exits the program. 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 evaluated 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 UNIX system 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 is reasonable if all bases 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.

; : Used by *bc*(1) for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1), hc(1)

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.

[This page left blank.]

NAME

deroff - remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [**-mx**] [**-w**] [files]

DESCRIPTION

The *deroff* command reads each of the *files* in sequence and removes all *troff*(1) requests, macro calls, backslash constructs, *eqn*(1) constructs (between **.EQ** and **.EN** lines and between delimiters), and *tbl*(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output.

The *deroff* command follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The **-m** option may be followed by an **m**, **s**, or **l**. The **-mm** option causes the macros to be interpreted so that only running text is output (i.e., no text from macro lines). The **-ml** option forces the **-mm** option and also causes deletion of lists associated with the **mm** macros.

If the **-w** option is given, the output is a word list, one word per line, with all other characters deleted. Otherwise, the output follows the original with the deletions mentioned above. In text, a word is any string that contains at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a word is a string that begins with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from words.

BUGS

The *deroff* command is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most of these errors result in too much rather than too little output.

The **-ml** option does not handle nested lists correctly.

[This page left blank.]

NAME

diff - differential file comparator

SYNOPSIS

diff [**-efbh**] file1 file2

DESCRIPTION

The *diff* command 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 with the name *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 comes 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 recreates *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

bdiff(1), cmp(1), comm(1), ed(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

WARNINGS

Missing newline at end of file X

Indicates that the last line of file X did not have a newline. If the lines are different, they are flagged and output although the output seems to indicate they are the same.

BUGS

Editing scripts produced under the -e or -f option are naive about creating lines consisting of a single period (.).

NAME

diff3 - 3-way differential file comparison

SYNOPSIS

diff3 [**-ex3**] file1 file2 file3

DESCRIPTION

The *diff3* command compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* 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 file is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

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 incorporates into *file1* 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 *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1)

BUGS

Text lines that consist of a single . defeat **-e**.
Files longer than 64K bytes do not work.

NAME

dircmp - directory comparison

SYNOPSIS

dircmp [-d] [-s] [-w n] [-o] dir1 dir2

DESCRIPTION

The *dircmp* command examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

-d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

-s Suppress messages about identical files.

-w*n* Change the width of the output line to *n* characters. The default width is 72.

-o Ignore time and date stamp differences when comparing the contents of binary files.

SEE ALSO

cmp(1), diff(1)

[This page left blank.]

NAME

echo - echo arguments

SYNOPSIS

echo [ ] [arg] ...

DESCRIPTION

The *echo* command writes its arguments separated by blanks and terminated by a newline on the standard output. The **-n** option prints a line without the newline; same as using the \c escape sequence.

The *echo* command also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b	backspace
\c	print line without newline
\f	form-feed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
****	backslash
\0*n*	where *n* is the 8-bit character whose ASCII code is the one, two or three digit octal number representing that character.

The *echo* command is useful for producing diagnostics in command files and for sending known data into a pipe.

SEE ALSO

sh(1)

CAVEATS

When representing an 8-bit character by using the escape convention **\0*n***, the *n* must **always** be preceded by the digit zero (0).

For example, typing: **echo 'WARNING:\07'** prints the phrase **WARNING:** and sounds the bell on your terminal. The use of single (or double) quotes (or two backslashes) is required to protect the \ that precedes the 07.

For the octal equivalents of each character, see *ascii*(5) in the Programmer's Reference Manual.

[This page left blank.]

NAME

ed, red - text editor

SYNOPSIS

ed [-s] [-p string] [-x] [-C] [file]
red [-s] [-p string] [-x] [-C] [file]

DESCRIPTION

The *ed* command is the standard text editor. If the *file* argument is given, *ed* simulates an *e* command (see the following text) on the named file; that is to say, the file is read into *ed*'s buffer so that it can be edited.

-s Suppresses the printing of character counts by *e*, *r*, and *w* commands, of diagnostics from *e* and *q* commands, and of the ! prompt after a *!shell command*.

-p Allows the user to specify a prompt string.

-x Encryption option; when used, *ed* simulates an **X** command and prompts the user for a key. This key is used to encrypt and decrypt text using the algorithm of *crypt*(1). The **X** command makes an educated guess to determine whether text read in is encrypted or not. The temporary buffer file is encrypted also, using a transformed version of the key typed in for the **-x** option. See *crypt*(1). Also, see the **WARNINGS** section at the end of this manual page.

-C Encryption option; the same as the **-x** option, except that *ed* simulates a **C** command. The **C** command is like the **X** command, except that all text read in is assumed to have been encrypted.

The *ed* command operates on a copy of the file it is editing; changes made to 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*. There is only one buffer.

The *red* command is a restricted version of *ed*. It allows editing of files only in the current directory. It prohibits executing shell commands by the *!shell command*. Attempts to bypass these restrictions result in an error message (*restricted shell*).

Both *ed* and *red* support the *fspec*(4) formatting capability. After including a format specification as the first line of *file* and invoking *ed* with your terminal in **stty -tabs** or **stty tab3** mode [see *stty*(1)], the specified tab stops are used automatically when scanning *file*. For example, if the first line of a file contained:

<:t5,10,15 s72:>

tab stops are set at columns 5, 10, and 15, and a maximum line length of 72 is imposed. NOTE: When you are entering text into the file, this format is not in effect; instead, because of being in **stty -tabs** or **stty tab3** mode, tabs are expanded to every eighth column.

Commands to *ed* have a simple and regular structure; zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in 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. Leave input mode by typing a period (.) at the beginning of a line, followed immediately by a carriage return.

The *ed* command supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be matched by the RE. The REs allowed by *ed* are constructed as follows:

The following one-character REs match a single character:

1.1 An ordinary character (not one of those discussed in 1.2 below) is a one-character RE that matches itself.

1.2 A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets ([]; see 1.4 in this list).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 in later text) or when it immediately follows the left of a pair of square brackets ([]) (see 1.4 in later text).

c. $ (dollar sign), which is special at the end of an entire RE (see 3.2 in this list).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE [for example, see how slash (/) is used in the *g* command, in the following text.]

1.3 A period (.) is a one-character RE that matches any character except newline.

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches any one character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character except newline and the remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (-) may be used to indicate a range of consecutive ASCII characters; for example, **[0-9]** is equivalent to **[0123456789]**. The - loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); e.g., **[ ]a-f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches zero or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a RE that matches a range of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256; \{*m*\} matches exactly *m* occurrences; \{*m*,\} matches at least *m* occurrences; \{*m*,*n*\} matches any number of occurrences between *m* and *n* inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 A RE enclosed between the character sequences \(and \) is a RE that matches whatever the unadorned RE matches.

2.6 The expression *n*, matches the same string of characters as was matched by an expression enclosed between \(and \) earlier in the same RE. Here

n is a digit; the sub-expression specified is that beginning with the *n*th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an entire RE may be constrained to match only an initial segment or final segment of a line (or both).

3.1 A circumflex (^) at the beginning of an entire RE constrains that RE to match an *initial* segment of a line.

3.2 A dollar sign ($) at the end of an entire RE constrains that RE to match a *final* segment of a line.

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (e.g., //) is equivalent to the last RE encountered. See also the last paragraph before **FILES.**

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; the exact effect on the current line is discussed under the description of each command. The *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 mark name character *x*, which must be an ASCII lowercase letter (a-z). Lines are marked with the *k* command described in the following text.
5. A RE enclosed by slashes (/) addresses the first line found by searching forward from the line following the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and including the current line, so that the entire buffer is searched. See also the last paragraph before **FILES.**
6. A RE enclosed in question marks (?) addresses the first line found by searching backward from the line preceding the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. See also the last paragraph before **FILES.**

7. An address followed by a plus sign (+) or a minus sign (-) followed by a decimal number specifies that address plus (respectively 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 to or subtracted from the address, respectively. As a consequence of this rule and of Rule 8, the address - refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to -.) Moreover, trailing + and - characters have a cumulative effect, so -- refers to the current line less 2.

10. For convenience, a comma (,) stands for the address pair **1,$**, while a semicolon (;) stands for the pair **.,$**.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) is used.

Typically, addresses are separated from each other by a comma (,). They may also be separated by a semicolon (;). In the latter case, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see Rules 5 and 6 shown previously). The second address of any two-address sequence must correspond to a line that follows, in the buffer, 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; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. However, any command (except *e*, *f*, *r*, or *w*) may be suffixed by **l**, **n**, or **p** in which case the current line is either listed, numbered, or printed, respectively, as discussed under the *l*, *n*, and *p* commands.

(.)a
<text>
.

The *a*ppend command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the

addressed line. Address 0 is legal for this command: it causes the appended text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text that replaces these lines; the . is left at the last line input, or, if there were none, at the first line that was not deleted.

(.,.)d

The *d*elete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

e *file*

The *e*dit command causes the entire contents of the buffer to be deleted, and then the named file to be read in; the . is set to the last line of the buffer. If no filename is given, the currently remembered filename, if any, is used (see the *f* command). The number of characters read is typed; *file* is remembered for possible use as a default filename in subsequent *e*, *r*, and *w* commands. If *file* is replaced by !, the rest of the line is taken to be a shell [*sh*(1)] command whose output is to be read. Such a shell command is not remembered as the current filename. See also **DIAGNOSTICS.**

E *file*

The *E*dit command is like *e*, except that the editor does not check to see if any changes have been made to the buffer since the last *w* command.

f *file*

If *file* is given, the *f*ile-name command changes the currently remembered filename to *file*; otherwise, it prints the currently remembered filename.

(1,$)g/*RE/command list*

In the *g*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with . initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a \; *a*, *i*, and *c* commands and associated input is permitted. The . which terminates the input mode may be omitted if it is the last line of the *command list*. An

empty *command list* is equivalent to the *p* command. The *g*, *G*, *v*, and *V* commands are not permitted in the *command list*. See also **BUGS** and the last paragraph before **FILES**.

(1,$)G/*RE***/**

In the interactive *G*lobal command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, . is changed to that line, and any *one* command (other than one of the *a*, *c*, *i*, *g*, *G*, *v*, and *V* commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a newline acts as a null command; an & causes the re-execution of the most recent command executed within the current invocation of *G*. Note that the commands input as part of the execution of the *G* command may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

h

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

H

The *H*elp command causes *ed* to enter a mode in which error messages are printed for all subsequent ? diagnostics. It will also explain the previous ? if there was one. The *H* command alternately turns this mode on and off; it is initially off.

(.)i
<text>
.

The *i*nsert command inserts the given text before the addressed line; the . is left at the last inserted line, or, if there was not one, at the addressed line. This command differs from the *a* command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate newline characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be an ASCII lowercase letter (**a-z**). The address '*x* then addresses this line;

the . is unchanged.

(.,.)l
The *l*ist command prints the addressed lines in an unambiguous way; a few non-printing characters (e.g., *tab, backspace*) are represented by visually mnemonic overstrikes. All other non-printing characters are printed in octal, and long lines are folded. An *l* command may be appended to any command other than *e*, *f*, *r*, or *w*.

(.,.)m*a*
The *m*ove command repositions the addressed line(s) after the line addressed by *a*. Address 0 is legal for *a* and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address *a* falls within the range of moved lines; . is left at the last line moved.

(.,.)n
The *n*umber command prints the addressed lines, preceding each line by its line number and a tab character; the . is left at the last line printed. The *n* command may be appended to any command other than *e*, *f*, *r*, or *w*.

(.,.)p
The *p*rint command prints the addressed lines; the . is left at the last line printed. The *p* command may be appended to any command other than *e*, *f*, *r*, or *w*. For example, *dp* deletes the current line and prints the new current line.

P
The editor prompts with a * for all subsequent commands. The *P* command alternately turns this mode on and off; it is initially off.

q
The *q*uit command causes *ed* to exit. No automatic write of a file is done; however, see **DIAGNOSTICS.**

Q
The editor exits without checking if changes have been made in the buffer since the last *w* command.

($)r *file*
The *r*ead command reads in the given file after the addressed line. If no filename is given, the currently remembered filename, if any, is used (see *e* and *f* commands). The currently remembered filename is not changed unless *file* is the very first filename mentioned since *ed* was invoked. 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 set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a shell [*sh*(1)] command whose output is to be read. For example, "$r !ls" appends current directory to the end of the file being edited. Such a shell command is not remembered as the current filename.

(.,.)s/*RE*/*replacement*/ or
(.,.)s/*RE*/*replacement*/g or
(.,.)s/*RE*/*replacement*/*n*

The *s*ubstitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (not overlapped) matched strings are replaced by the *replacement* 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. If a number *n* appears after the command, only the *n*th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or newline may be used instead of / to delimit the RE and the *replacement*; . is left at the last line on which a substitution occurred. See also the last paragraph before **FILES**.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of & in this context may be suppressed by preceding it by \. As a more general feature, the characters *n*, where *n* is a digit, are replaced by the text matched by the *n*th regular subexpression of the specified RE enclosed between \(and \). When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

A line may be split by substituting a newline character into it. The newline in the *replacement* must be escaped by preceding it with \. Such substitution cannot be done as part of a *g* or *v* command list.

(.,.)t*a*

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); the . is left at the last line of the copy.

u

The *u*ndo command nullifies the effect of the most recent command that modified anything in the buffer, namely the most recent *a*, *c*, *d*, *g*, *i*, *j*, *m*, *r*, *s*, *t*, *v*, *G*, or *V* command.

(1,$)v/*RE*/*command list*

This command is the same as the global command *g* except that the *command list* is executed with . initially set to every line that does not match the RE.

(1,$)V/*RE*/

This command is the same as the interactive global command *G* except that the lines that are marked during the first step are those that do not match the RE.

(1,$)w *file*

The *w*rite command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your *umask* setting [see *umask*(1)] dictates otherwise. The currently remembered filename is not changed unless *file* is the very first filename mentioned since *ed* was invoked. If no filename is given, the currently remembered filename, if any, is used (see *e* and *f* commands); the . is unchanged. If the command is successful, the number of characters written is typed. If *file* is replaced by !, the rest of the line is taken to be a shell [*sh*(1)] command whose standard input is the addressed lines. Such a shell command is not remembered as the current filename.

X

A key is prompted for, and it is used in subsequent *e*, *r*, and *w* commands to decrypt and encrypt text using the *crypt*(1) algorithm. An educated guess is made to determine whether text read in for the *e* and *r* commands is encrypted. A null key turns off encryption. Subsequent *e*, *r*, and *w* commands will use this key to encrypt or decrypt the text [see *crypt*(1)]. An explicitly empty key turns off encryption. Also, see the **-x** option of *ed*.

(**$**)**=**

The line number of the addressed line is typed; the . is unchanged by this command.

!*shell command*

The remainder of the line after the ! is sent to the UNIX system shell [*sh*(1)] to be interpreted as a command. Within the text of that command, the unescaped character **%** is replaced with the remembered filename; if a ! appears as the first character of the shell command, it is replaced with the

text of the previous shell command. Thus, !! will repeat the last shell command. If any expansion is performed, the expanded line is echoed; the . is unchanged.

(.+1)<newline>
An address alone on a line causes the addressed line to be printed. A newline alone is equivalent to .+1p; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII DEL or BREAK) is sent, *ed* prints a ? and returns to its command level.

Some size limitations: 512 characters in a line, 256 characters in a global command list, and 64 characters in the pathname of a file (counting slashes). The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

When reading a file, *ed* discards ASCII NUL characters.

If a file is not terminated by a newline character, *ed* adds one and puts out a message explaining what it did.

If the closing delimiter of a RE or of a replacement string (e.g., /) is the last character before a newline, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

s/s1/s2	s/s1/s2/p
g/s1	g/s1/p
?s1	?s1?

FILES

$TMPDIR	If this environmental variable is not null, its value is used in place of **/usr/tmp** as the directory name for the temporary work file.
/usr/tmp	If **/usr/tmp** exists, it is used as the default directory name for the temporary work file.
/tmp	If the environmental variable **TMPDIR** does not exist or is null, and if **/usr/tmp** does not exist, then **/tmp** is used as the directory name for the temporary work file.
ed.hup	Work is saved here if the terminal is hung up.

NOTES

The - option, although it continues to be supported, has been replaced in the documentation by the -s option that follows the Command Syntax Standard [see *intro*(1)]. The - option will not be supported in the next major release of the operating system.

SEE ALSO

edit(1), ex(1), grep(1), sed(1), sh(1), stty(1), umask(1), vi(1)
fspec(4), regexp(5) in the Programmer's Reference Manual

DIAGNOSTICS

?	for command errors
?file	for an inaccessible file (use *h*elp)

If changes have been made in the buffer since the last *w* command that wrote the entire buffer, *ed* warns the user if an attempt is made to destroy *ed*'s buffer by the *e* or *q* commands. It prints ? and allows one to continue editing. A second *e* or *q* command at this point will take effect. The -s command-line option inhibits this feature.

WARNINGS

The encryption options and commands are provided with the Crypt Utilities package, which is available only in the United States.

BUGS

A *!* command cannot be subject to a *g* or a *v* command.

The *!* command and the ! escape from the *e*, *r*, and *w* commands cannot be used if the editor is invoked from a restricted shell [see *sh*(1)].

The sequence \n in a RE does not match a newline character.

If the editor input is coming from a command file (e.g., ed file < ed-cmd-file), the editor will exit at the first failure.

NAME

edit - text editor (variant of ex for casual users)

SYNOPSIS

edit [-r] [-x] [-C] name ...

DESCRIPTION

The *edit* command is a variant of the text editor *ex*, recommended for new or casual users who wish to use a command-oriented editor. It operates precisely as *ex*(1) with the following options automatically set:

novice	ON
report	ON
showmode	ON
magic	OFF

These options can be turned on or off with the *set* command in *ex*(1).

-r Recover file after an editor or system crash.

-x Encryption option; when used, the file will be encrypted as it is being written and will require an encryption key to be read. The *edit* command makes an educated guess to determine if a file is encrypted or not. See *crypt*(1). Also, see the **WARNINGS** section at the end of this manual page.

-C Encryption option; the same as -x except that *edit* assumes files are encrypted.

The following brief introduction should help you get started with *edit*. If you are using a CRT terminal you may want to learn about the display editor *vi*.

To edit the contents of an existing file, you begin with the command **edit** *name* to the shell. The *edit* command makes a copy of the file that you can then edit, and tells you how many lines and characters are in the file. To create a new file, you also begin with the command **edit** with a filename, **edit** *name*; the editor tells you it is a `New File`.

The *edit* command prompt is the colon (:), which you should see after starting the editor. If you are editing an existing file, then you have some lines in *edit*'s buffer (its name for the copy of the file you are editing). When you start editing, *edit* makes the last line of the file the current line. Most commands to *edit* use the current line if you do not tell them which line to use. Thus if you say *print* (which can be abbreviated *p*) and type carriage return (as you should after all *edit* commands), the current line is printed. If you *delete* (*d*) the current line, *edit*

prints the new current line, which is usually the next line in the file. If you *delete* the last line, then the new last line becomes the current one.

If you start with an empty file or wish to add some new lines, then the *append* (*a*) command can be used. After you execute this command (typing a carriage return after the word *append*), *edit* will read lines from your terminal until you type a line consisting of just a dot (.); it places these lines after the current line. The last line you type then becomes the current line. The command *insert* (*i*) is like *append*, but places the lines you type before, rather than after, the current line.

The *edit* command numbers the lines in the buffer, with the first line having number 1. If you execute the command *1*, then *edit* will type the first line of the buffer. If you then execute the command *d*, *edit* will delete the first line, line 2 will become line 1, and *edit* will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the *substitute* (*s*) command, s/*old*/*new*/ where *old* is the string of characters you want to replace and *new* is the string of characters you want to replace *old* with.

The command *file* (*f*) will tell you how many lines there are in the buffer you are editing and will say `[Modified]` if you have changed the buffer. After modifying a file, you can save the contents of the file by executing a *write* (*w*) command. You can leave the editor by issuing a *quit* (*q*) command. If you run *edit* on a file, but do not change it, it is not necessary (but does no harm) to *write* the file back. If you try to *quit* from *edit* after modifying the buffer without writing it out, you will receive the message `No write since last change (:quit! overrides)`, and *edit* will wait for another command. If you do not want to write the buffer out, issue the *quit* command followed by an exclamation point (*q!*). The buffer is then irretrievably discarded and you return to the shell.

By using the *d* and *a* commands and giving line numbers to see lines in the file, you can make any changes you want. You should learn at least a few more things, however, if you will use *edit* more than a few times.

The *change* (*c*) command changes the current line to a sequence of lines you supply as in *append*, you type lines up to a line consisting of only a dot (.). You can tell *change* to change more than one line by giving the line numbers of the lines you want to change, i.e., **3,5c**. You can print lines this way too, **1,23p** prints the first 23 lines of the file.

The *undo* (*u*) command reverses the effect of the last command you executed that changed the buffer. Thus, if you execute a *substitute* command that does not do what you want, type *u* and the old contents of the line will be restored. You can also undo an *undo* command. The *edit* command will give you a warning message when a command affects more than one line of the buffer. Note that commands such as *write* and *quit* cannot be undone.

To look at the next line in the buffer, type carriage return. To look at a number of lines, type **^D** (while holding down the control key, press **d**) rather than carriage return. This will show you a half-screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at nearby text by executing the *z* command. The current line will appear in the middle of the text displayed, and the last line displayed will become the current line; you can get back to the line where you were before you executed the *z* command by typing . The *z* command has other options, *z-* prints a screen of text (or 24 lines) ending where you are; *z+* prints the next screenful. If you want less than a screenful of lines, type *z.11* to display five lines before and five lines after the current line. (Typing **z***n*, when *n* is an odd number, displays a total of *n* lines, centered about the current line; when *n* is an even number, it displays *n*-1 lines, so that the lines displayed are centered around the current line.) You can give counts after other commands; for example, you can delete 5 lines starting with the current line with the command **d5**.

To find things in the file, you can use line numbers if you happen to know them. Since the line numbers change when you insert and delete lines, this is somewhat unreliable. You can search backward and forward in the file for strings by giving commands of the form */text/* to search forward for *text* or *?text?* to search backward for *text*. If a search reaches the end of the file without finding *text*, it wraps around and continues to search back to the line where you are. A useful feature here is a search of the form */^text/* which searches for *text* at the beginning of a line. Similarly */text$/* searches for *text* at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has the symbolic name dot (.); this is most useful in a range of lines as in **.,$p** which prints the current line plus the rest of the lines in the file. To move to the last line in the file, you can refer to it by its symbolic name $. Thus the command **$d** deletes the last line in the file, no matter what the current line is. Arithmetic with line references is also possible. Thus the line **$-5** is the fifth before the last and **.+20** is 20 lines after the current line.

You can determine the current line by typing **.=** . This is useful if you wish to move or copy a section of text within a file or between files. Find the first and last line numbers you wish to copy or move. To move lines 10 through 20, type

10,20d a to delete these lines from the file and place them in a buffer named **a**. The *edit* command has 26 such buffers named **a** through **z**. To put the contents of buffer **a** after the current line, type *put a*. If you want to move or copy these lines to another file, execute an *edit* (*e*) command after copying the lines; following the *e* command with the name of the other file you wish to edit, i.e., *edit chapter2*. To copy lines without deleting them, use *yank* (*y*) in place of **d**. If the text you wish to move or copy is all within one file, it is not necessary to use named buffers. For example, to move lines 10 through 20 to the end of the file, type **10,20m $.**

SEE ALSO

ed(1), ex(1), vi(1)

WARNING

The encryption options are provided with the Crypt Utilities package, which is available only in the United States.

NAME

egrep - search a file for a pattern using full regular expressions

SYNOPSIS

egrep [options] full regular expression [file ...]

DESCRIPTION

The *egrep* command (*expression grep*) searches files for a pattern of characters and prints all lines that contain that pattern. The *egrep* command uses full regular expressions (expressions that have string values that use the full set of alphanumeric and special characters) to match the patterns. It uses a fast deterministic algorithm that sometimes needs exponential space.

The *egrep* command accepts full regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A full regular expression followed by + that matches one or more occurrences of the full regular expression.
2. A full regular expression followed by ? that matches 0 or 1 occurrences of the full regular expression.
3. Full regular expressions separated by ¦ or by a newline that match strings that are matched by any of the expressions.
4. A full regular expression that may be enclosed in parentheses () for grouping.

Be careful using the characters $, *, [, ^, ¦, (,), and \ in *full regular expression*, because they are also meaningful to the shell. It is safest to enclose the entire *full regular expression* in single quotes '...'.

The order of precedence of operators is [ ], then *? +, then concatenation, then ¦ and newline.

If no files are specified, *egrep* assumes standard input. Normally, each line found is copied to the standard output. The filename is printed before each line found if there is more than one input file.

Command line options are:

-b Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).

-c Print only a count of the lines that contain the pattern.

-i Ignore upper/lower case distinction during comparisons.

-l Print the names of files with matching lines once, separated by newlines. Does not repeat the names of files when the pattern is found more than once.

-n Precede each line by its line number in the file (first line is 1).

-v Print all lines except those that contain the pattern.

-e *special_expression*
Search for a *special expression* (*full regular expression* that begins with a -).

-f *file* Take the list of *full regular expressions* from *file*.

-h Prevents the name of the file containing the matching lines from being appended to that line. Used when searching multiple files.

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 (even if matches were found).

BUGS

Ideally, there should be only one *grep* command, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h**, which is included as part of the basic software development set.

NAME

enable, disable - enable/disable LP printers

SYNOPSIS

enable *printers*
disable [*options*] *printers*

DESCRIPTION

The *enable* command activates the named *printers*, enabling them to print requests taken by *lp*(1). Use *lpstat*(1) to find the status of printers.

The *disable* command deactivates the named *printers*, disabling them from printing requests taken by *lp*(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use *lpstat*(1) to find the status of printers. Options for use with *disable* are:

-c Cancel any requests that are currently printing on any of the designated printers.

-r *reason* Assign a *reason* for the disabling of the printers. This *reason* applies to all printers mentioned up to the next **-r** option. This *reason* is reported by *lpstat*(1). If the **-r** option is not present, then a default reason will be used.

FILES

/usr/spool/lp/*

SEE ALSO

lp(1), lpstat(1)

[This page left blank.]

NAME

env - set environment for command execution

SYNOPSIS

env [-] [name=value] ... [command args]

DESCRIPTION

The *env* command obtains the current *environment*, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form *name* =*value* are merged into the inherited environment before the command is executed. The - flag causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

Internationalization

Characters from supplementary code sets can be used for *value*, *command* and *args*.

SEE ALSO

sh(1)

exec(2), profile(4), environ(5) in the Programmer's Reference Manual

[This page left blank.]

NAME

eqn, neqn, checkeq – format mathematical text for nroff or troff

SYNOPSIS

eqn [**–dxy**] [**–pn**] [**–sn**] [**–fn**] [files]

neqn [**–dxy**] [**–pn**] [**–sn**] [**–fn**] [files]

checkeq [files]

DESCRIPTION

Eqn is a *troff*(1) preprocessor for typesetting mathematical text on a phototypesetter, while *neqn* is used for the same purpose with *nroff* on typewriter-like terminals. Usage is almost always:

```
eqn  files  | troff
neqn files  | nroff
```

or equivalent.

If no files are specified (or if – is specified as the last argument), these programs read 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, and so forth. It is also possible to designate two characters as *delimiters*; subsequent text between delimiters is then treated as *eqn* 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 the same character; the dollar sign is often used as such a delimiter. 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, new-lines, braces, double quotes, tildes, and circumflexes. Braces { } are used for grouping; generally speaking, anywhere a single character such as *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.

Subscripts and superscripts are produced with the keywords **sub** and **sup**. Thus *x sub j* makes x_j, *a sub k sup 2* produces a_k^2, while $e^{x^2+y^2}$ is made with *e sup {x sup 2 + y sup 2}*.

Fractions are made with **over**: *a over b* yields $\frac{a}{b}$; **sqrt** makes square roots:

1 over sqrt {ax sup 2+bx+c} results in:

$$\frac{1}{\sqrt{ax^2+bx+c}}.$$

The keywords **from** and **to** introduce lower and upper limits:

$$\lim_{n\to\infty}\sum_{0}^{n}x_i$$

is made with *lim from {n -> inf} sum from 0 to n x sub i*. Left and right brackets, braces, and so forth, of the right height are made with **left** and **right**:

left [x sup 2 + y sup 2 over alpha right] ~=~ 1

produces

$$\left[x^2+\frac{y^2}{\alpha}\right]=1.$$

Legal characters after **left** and **right** are braces, brackets, bars, **c** and **f** for ceiling and floor, and "" for nothing at all (useful for a right-side-only bracket). A **left** *thing* need not have a matching **right** *thing*.

Vertical piles of things are made with **pile**, **lpile**, **cpile**, and **rpile**:

pile {a above b above c}

produces:

a
b.
c

Piles may have arbitrary numbers of elements; **lpile** left-justifies, **pile** and **cpile** center (but with different vertical spacing), and **rpile** right justifies. Matrices are made with **matrix**:

matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }

produces

x_i 1

y_2 2

In addition, there is **rcol** for a right-justified column.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **tilde**, **bar**, **vec**, **dyad**, and **under**: *x dot = f(t) bar* is $\dot{x}=\overline{f(t)}$, *y dotdot bar ~=~ n under* is $\bar{\ddot{y}}=\underline{n}$, and *x vec ~=~ y dyad* is $\overrightarrow{x}=\overleftrightarrow{y}$.

Point sizes and fonts can be changed with **size** *n* or **size** ±*n*, **roman**, **italic**, **bold**, and **font** *n*. Point sizes and fonts can be changed globally in a document by **gsize** *n* and

gfont *n*, or by the command-line arguments **–s***n* and **–f***n*.

Normally, subscripts and superscripts are reduced by 3 points from the previous size; this may be changed by the command-line argument **–p***n*.

Successive display arguments can be lined up. Place **mark** before the desired lineup point in the first equation; place **lineup** at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with **define**:

define thing % replacement %

defines a new token called *thing* that will be replaced by *replacement* whenever it appears thereafter. The % may be any character that does not occur in *replacement*.

Keywords such as **sum** (∑), **int** (∫), **inf** (∞), and shorthands such as >= (≥), != (≠), and -> (→) are recognized. Greek letters are spelled out in the desired case, as in **alpha** (α), or **GAMMA** (Γ). Mathematical words such as **sin**, **cos**, and **log** are made Roman automatically. *Troff*(1) four-character escapes such as \(**dd** (‡) and \(**bs** (\) may be used anywhere. Strings enclosed in double quotes ("...") are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *troff*(1) when all else fails. Full details are given in the manual cited below.

SEE ALSO

Documenter's Workbench User's Guide
AT&T Documenter's Workbench Technical Discussion and Reference

BUGS

To embolden digits, parentheses, and so forth, it is necessary to quote them, as in **bold "12.3"**.

See also "BUGS" under *troff*(1).

[This page left blank.]

NAME

eucset - set or get EUC code set width

SYNOPSIS

eucset [cswidth]
eucset -p

DESCRIPTION

The *eucset* command assumes the existence of an EUC line discipline (which does canonical processing of EUC characters) in its standard input Stream (usually a tty). The EUC line discipline must recognize the *eucioc* calls to *ioctl*(2), as defined in the header file **sys/eucioctl.h.**

If given no arguments, *eucset* looks in the environment for the **cswidth** parameter in the character class table, which is assumed to specify the code set widths and screen widths in use. The format of **cswidth** parameter is described in character class table specification.

If given one argument which does not begin with "_", it is taken to be a string in the format of **cswidth** parameter, overriding whatever is in the environment.

If given the optional argument **-p**, *eucset* prints the current values of the code set widths and Screen widths as returned by the line discipline. These values may be different than what is currently in the user's environment, but represents the EUC mapping that the EUC line discipline is currently using. Code set 0 (ASCII) is excluded from the listing, which is in the same format as the **cswidth** parameter.

RETURN VALUES

The *eucset* command returns 0 on success, 1 on failure of any call to *ioctl*(2).

FILES

/usr/include/sys/eucioctl.h
/usr/include/sys/euc.h
/usr/include/sys/cswidth.h

SEE ALSO

ioctl(2), getwidth(3W), eld0(7), streamio(7) in the Programmer's Reference Manual

[This page left blank.]

NAME

ex - text editor

SYNOPSIS

ex [-s] [-v] [-t tag] [-r file] [-L] [-R] [-x] [-C] [-c command] file ...

DESCRIPTION

The *ex* command is the root of a family of editors: *ex* and *vi*. The *ex* command 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 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*.

For ed Users

If you have used *ed*(1), you will find that, in addition to having all of the *ed*(1) commands available, *ex* has a number of additional features useful on CRT terminals. Intelligent terminals and high-speed terminals are very pleasant to use with *vi*. Generally, the *ex* editor uses far more of the capabilities of terminals than *ed*(1) does and uses the terminal capability data base [see *terminfo* (4)] and the type of the terminal you are using from the environmental variable TERM 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).

The *ex* command contains a number of features for easily viewing the text of the file. The *z* command gives easy access to windows of text. Typing **^D** (control-d) causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just typing return. Of course, the screen-oriented *visual* mode gives constant access to editing context.

The *ex* command gives you help when you make mistakes. The *undo* (*u*) command allows you to reverse any single change which goes astray. The *ex* command 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 do not accidentally overwrite a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the telephone, you can use the editor *recover* command (or **-r** *file* option) to retrieve your work. This will get you back to within a few lines of where you left off.

The *ex* command 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 filenames 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.

The editor has a group of buffers whose names are the ASCII lowercase letters (**a-z**). You can place text in these named buffers where it is available to be inserted elsewhere in the file. The contents of these buffers remain available when you begin editing a new file using the *edit* (*e*) command.

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.

It is possible to ignore the case of letters in searches and substitutions. The *ex* command 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".

The *ex* command has a set of options which you can set to tailor your document to your liking. One option which is very useful is the **autoindent** option that allows the editor to supply leading white space to align text automatically. You can then use **^D** as a backtab and space or tab to move forward to align new code easily.

Miscellaneous useful features include an intelligent *join* (*j*) command that 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*(1).

Invocation Options

The following invocation options are interpreted by *ex* (previously documented options are discussed in the **NOTES** section at the end of this manual page):

-s	Suppress all interactive-user feedback. This is useful in processing editor scripts.
-v	Invoke *vi*.
-t *tag*	Edit the file containing the *tag* and position the editor at its definition.

-r *file*	Edit *file* after an editor or system crash. (Recovers the version of *file* that was in the buffer when the crash occurred.)
-L	List the names of all files saved as the result of an editor or system crash.
-R	Read only mode; the read only flag is set, preventing accidental overwriting of the file.
-x	Encryption option; when used, *ex* simulates an **X** command and prompts the user for a key. This key is used to encrypt and decrypt text using the algorithm of *crypt*(1). The **X** command makes an educated guess to determine whether text read in is encrypted or not. The temporary buffer file is encrypted also, using a transformed version of the key typed in for the **-x** option. [See *crypt*(1)]. Also, see the **WARNINGS** section at the end of this manual page.
-C	Encryption option; the same as the **-x** option, except that *ex* simulates a **C** command. The **C** command is like the **X** command, except that all text read in is assumed to have been encrypted.
-c *command*	Begin editing by executing the specified editor *command* (usually a search or positioning command).

The *file* argument indicates one or more files to be edited.

The ex States

Command	Normal and initial state. Input prompted for by :. Your line kill character cancels a partial command.
Insert	Entered by **a**, **i**, or **c**. Arbitrary text may be entered. Insert state normally is terminated by a line having only "." on it, or, abnormally, with an interrupt.
Visual	Entered by typing **vi**; terminated by typing **Q** or ^\ (control-\).

The ex Command Names and Abbreviations

abbrev	**ab**	**map**		**set**	**se**
append	**a**	**mark**	**ma**	**shell**	**sh**
args	**ar**	**move**	**m**	**source**	**so**
change	**c**	**next**	**n**	**substitute**	**s**
copy	**co**	**number**	**nu**	**unabbrev**	**unab**
delete	**d**	**preserve**	**pre**	**undo**	**u**

edit	**e**	**print**	**p**	**unmap**	**unm**
file	**f**	**put**	**pu**	**version**	**ve**
global	**g**	**quit**	**q**	**visual**	**vi**
insert	**i**	**read**	**r**	**write**	**w**
join	**j**	**recover**	**rec**	**xit**	**x**
list	**l**	**rewind**	**rew**	**yank**	**ya**

The ex Commands

forced encryption	C	heuristic encryption	X
resubst	&	print next	CR
rshift	>	lshift	<
scroll	^D	window	z
shell escape	!		

The ex Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	*?pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
-	previous	*'x*	marked with *x*
+*n*	*n* forward	''	previous context
%	1,$		

Initializing Options

EXINIT	place **set** here in environment variable
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option *x*
set no*x*	disable option *x*
set *x*=*val*	give value *val* to option *x*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Most Useful Options and Their Abbreviations

autoindent	**ai**	supply indent
autowrite	**aw**	write before changing files
directory	**dir**	specify the directory
exrc	**ex**	allow *vi*/*ex* to read the **.exrc** in the current directory. This option is set in the **EXINIT** shell variable or in the **.exrc** file in the **$HOME** directory.
ignorecase	**ic**	ignore case of letters in scanning

list		print ^I for tab, $ at end
magic		treat . [* special in patterns
modelines		first five lines and last five lines executed as *vi/ex* commands if they are of the form **ex:***command*: or **vi:***command*:
number	**nu**	number lines
paragraphs	**para**	macro names that start paragraphs
redraw		simulate smart terminal
report		informs you if the number of lines modified by the last command is greater than the value of the **report** variable
scroll		command mode lines
sections	**sect**	macro names that start sections
shiftwidth	**sw**	for < >, and input ^D
showmatch	**sm**	to) and } as typed
showmode	**smd**	show insert mode in *vi*
slowopen	**slow**	stop updates during insert
term		specifies to **vi** the type of terminal being used (the default is the value of the environmental variable **TERM**)
window		visual mode lines
wrapmargin	**wm**	automatic line splitting
wrapscan	**ws**	search around end (or beginning) of buffer

Scanning Pattern Formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any character in *str*
[↑*str*]	any character not in *str*
[*x*-*y*]	any character between *x* and *y*
*	any number of preceding characters

Internationalization

The *ex* command can process characters from supplementary code sets as well as ASCII characters.

In regular expressions, searches and pattern matching are performed in character units, not in individual bytes.

FILES

/usr/lib/exstrings	error messages
/usr/lib/exrecover	recover command
/usr/lib/expreserve	preserve command
/usr/lib/terminfo/*	describes capabilities of terminals
$HOME/.exrc	editor startup file
./.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/R*xnnnnn*	named buffer temporary
/usr/preserve/*login*	preservation directory (where *login* is the user's login)

NOTES

Several options, although they continue to be supported, have been replaced in the documentation by options that follow the Command Sytax Rules [see intro(1)]. The - option has been replaced by -s, a -r option that is not followed with an option-argument has been replaced by -L, and +*command* has been replaced by -c *command*.

SEE ALSO

crypt(1), ed(1), edit(1), grep(1), sed(1), sort(1), vi(1)
curses(3X), term(4), terminfo(4) in the Programmer's Reference Manual
User's Guide
"curses/terminfo" chapter of the Programmer's Guide

WARNINGS

The encryption options and commands are provided with the Crypt Utilities package, which is available only in the United States.

BUGS

The *z* command prints the 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 do not print a name if the command line -s option is used.

There is no easy way to do a single scan ignoring case.

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

expand, unexpand - expand tabs to spaces, and vice versa

SYNOPSIS

expand [-tabstop] [-tab1, tab2, ..., tabn] [file ...]

unexpand [**-a**] [file ...]

DESCRIPTION

The *expand* command processes the named files or the standard input writing the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. Use *expand* for pre-processing character files (before sorting, looking at specific columns, and so forth) that contain tabs.

If a single *tabstop* argument is given then tabs are set *tabstop* spaces apart instead of the default 8. If multiple tabstops are given then the tabs are set at those specific columns.

Use *unexpand* to put tabs back into the data from the standard input or the named files and to write the result on the standard output. By default, only leading blanks and tabs are reconverted to maximal strings of tabs. If the **-a** option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

[This page left blank.]

NAME

expr - evaluate arguments as an expression

SYNOPSIS

expr arguments

DESCRIPTION

The *arguments* are taken as an expression. After evaluation, the result is written on the standard output. Terms of the expression must be separated by blanks. Characters special to the shell must be escaped. Note that **0** is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2s complement numbers.

The operators and keywords are listed in the following text. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr \| *expr*
: returns the first *expr* if it is neither null nor **0**, otherwise returns the second *expr*.

expr \& *expr*
: returns the first *expr* if neither *expr* is null or **0**, otherwise returns **0**.

expr { =, \>, \> =, \<, \< =, != } *expr*
: returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

expr { +, - } *expr*
: addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
: multiplication, division, or remainder of the integer-valued arguments.

expr : *expr*
: The matching operator : compares the first argument with the second argument which must be a regular expression. Regular expression syntax is the same as that of *ed*(1), except that all patterns are anchored (i.e., begin with ^) therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (**0** on failure). Alternatively, the \(...\) pattern symbols can be used to return a portion of the first argument.

EXAMPLES

1. a= ` expr $a + 1 `

   adds 1 to the shell variable a.

2. # 'For $a equal to either "/usr/abc/file" or just "file" '
   expr $a : '.*/\(.*\)' \| $a

   returns the last segment of a pathname (i.e., **file**). Watch out for / alone as an argument; *expr* will take it as the division operator (see "BUGS" at the end of this manual page).

3. # A better representation of example 2:
   expr //$a : '.*/\(.*\)

   The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. expr $VAR : '.*'

   returns the number of characters in **$VAR**.

SEE ALSO

ed(1), sh(1)

DIAGNOSTICS

As a side effect of expression evaluation, *expr* returns the following exit values;

0	if the expression is neither null nor **0**
1	if the expression *is* null or **0**
2	for invalid expressions
syntax error	for operator/operand errors
non-numeric argument	if arithmetic is attempted on such a string

BUGS

After argument processing by the shell, *expr* cannot tell the difference between an operator and an operand except by the value. If **$a** is an =, the command **expr $a = '='** looks like **expr = = =** as the arguments are passed to *expr* (and they will all be taken as the = operator). The following works:
expr X$a = X=

NAME

exstr - extract strings from source files

SYNOPSIS

exstr files
exstr *-e* files
exstr *-r* [*-d*] file

DESCRIPTION

The *exstr* utility is used to extract strings from source files and replace them by calls to the message retrieval function [*gettxt*(3G)]. In the first form, *exstr* finds all strings in the source files and writes them on the standard output. Each string is preceded by the source filename and a colon. The meanings of the options are:

-e

Produces on the standard output a list of strings from the named C source files with positional information.

The following is the format of the output:

file:line:position:msgfile:msgnum:*string*

The following fields are created by *exstr*:

file	the name of the C source file
line	line number in the file
position	character position in the line
string	the extracted text string

The application developer will supply the remaining two fields:

msgfile	the file that contains the created text strings

A file with the same name must be created and installed in the appropriate place by the *mkmsgs*(1M) utility.

msgnum	the sequence number of the string in the message file

This file must be examined by the developer to identify which strings can be translated and retrieved by the message retrieval function. The developer has to delete lines that cannot be translated and insert the names of message file(s) and message number(s) in their appropriate places. The names of message files must be the same as the names of files created by *mkmsgs*(1M) and installed in

/lib/locale/<*locale*>/LC_MESSAGES. The message numbers must correspond to the sequence numbers of strings in the message files.

-r Modify the C source file by replacing strings with function calls to the message retrieval function *gettxt*.

The first step is to invoke *exstr* with the **-e** option to create a list of strings. This list will be examined and modified by deleting lines and adding the message filenames and numbers to their appropriate places. The *exstr* command with the **-r** option will use the modified list of strings as input C source file. Strings in the source file are replaced by call to the message retrieval function *gettxt*(3G). The *msgfile* and *msgnum* fields will be used to construct the first argument to *gettxt*. Without the **-d** option the second argument to *gettxt* will be the null string [see *gettxt*(3G)].

-d This option is used together with the **-r** option. The extracted strings are used as second arguments to the retrieval function (*gettxt*).

This utility would not be capable of performing strings replacement in all instances. For example, a static initialized character string cannot be replaced by a function call. Another example is a string which could be in a form of an escape sequence which would not be translated. These examples strongly suggest that, in order not to break existing code, the files created by invoking *exstr* with the **-e** option must be examined, and lines containing strings not replaceable by function calls must be deleted. In some cases, the code may require modifications so that strings can be extracted and replaced by calls to the message retrieval function.

EXAMPLES

The following examples show uses of *exstr*.

The file **foo.c** contains two strings.

```
main()
{
        printf(:This is an example\n");
        printf("Hello world!!\n");
}
```

The *exstr* utility, invoked with the argument **foo.c** extracts strings from the named file and prints them on the standard output.

exstr foo.c will produce the following output:

```
foo.c:This is an example\n
foo.c:Hello world\n
```

exstr -e foo.c will produce the following output:

```
foo.c:3:8:::This is an example\n
foo.c:4:8:::Hello world!\n
```

The developer must supply the *msgfile* and *msgnum* fields before the strings can be replaced by calls to the retrieval function. If *UX* is the name of the message file and the numbers *1* and *2* represent the sequence number of the strings in the file the following are the contents of strings file after this information has been added:

foo.c:3:8:UX:1:This is an example\n
foo.c:4:8:UX:2:Hello world!\n

The *exstr* utility can now be invoked with the **-r** option to replace the strings in the source file by calls to the message retrieval function (*gettxt*).

exstr -r foo.c < strings > intlfoo.c

The following is the output of the command:

```
extern char *gettxt():
main()
{
        printf(gettxt("UX:1",""));
        printf(gettxt("UX:2","");
}
```

The exstr -rd foo.c < strings > intlfoo.c will use the extracted strings as a second argument to *2gettxt*.

```
extern char *gettxt();
main()
{
        printf(gettxt("UX:1","This is an example\n"));
        printf(gettxt("UX:2","Hello world!!\n"));
```

FILES

/lib/locale/<locale>/LC_MESSAGES/*
message file created by *mkmsgs*(1M)

SEE ALSO

mkmsgs(1M) in the Administrator's Reference Manual
setlocale(3C), gettxt(3G) in the Programmer's Reference Manual

DIAGNOSTICS

The error messages produced by *exstr* are intended to be self-explanatory. They indicate errors in the command line or format errors encountered within the strings file.

NAME

factor – obtain the prime factors of a number

SYNOPSIS

factor [integer]

DESCRIPTION

When you use *factor* without an argument, it waits for you to give it an integer. After you give it a positive integer less than or equal to 10^{14}, it factors the integer, prints its prime factors the proper number of times, and then waits for another integer. The *factor* command exits if it encounters a zero or any non-numeric character.

If you invoke *factor* with an argument, it factors the integer as described above, and then it exits.

The maximum time to factor an integer is proportional to $\sqrt{n}$. The *factor* command will take this time when n is prime or the square of a prime.

DIAGNOSTICS

The *factor* command prints the error message, `Ouch`, for input out of range or for garbage input.

[This page left blank.]

NAME

fgrep - search a file for a character string

SYNOPSIS

fgrep [options] string [file ...]

DESCRIPTION

The *fgrep* (fast *grep*) command searches files for a character string and prints all lines that contain that string. The *fgrep* command is different from *grep(1)* and *egrep(1)* because it searches for a string, instead of searching for a pattern that matches an expression. It uses a fast and compact algorithm.

The characters $, *, [, ^, |, (,), and \ are interpreted literally by *fgrep*, that is, *fgrep* does not recognize full regular expressions as does *egrep*. Since these characters have special meaning to the shell, it is safest to enclose the entire *string* in single quotes '. . .'.

If no files are specified, *fgrep* assumes standard input. Normally, each line found is copied to the standard output. The filename is printed before each line found, if there is more than one input file.

Command line options are:

- **-b** Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).
- **-c** Print only a count of the lines that contain the pattern.
- **-i** Ignore upper/lower case distinction during comparisons.
- **-l** Print the names of files with matching lines once, separated by newlines. Does not repeat the names of files when the pattern is found more than once.
- **-n** Precede each line by its line number in the file (first line is 1).
- **-v** Print all lines except those that contain the pattern.
- **-x** Print only lines matched entirely.
- **-e** *special_string*
  Search for a *special string* (*string* begins with a -).
- **-f** *file* Take the list of *strings* from *file*.
- **-h** Prevents the name of the file containing the matching line from being appended to that line. Used when searching multiple files.

Internationalization

The *fgrep* command can process characters from supplementary code sets.

Searches are performed on characters, not on individual bytes.

Option:

-i Ignore upper/lowercase distinction during comparisons; valid for single-byte characters only.

SEE ALSO

ed(1), egrep(1), grep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, and 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *grep* command, but there is not a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h**, which is included as part of the basic software development set.

NAME

file - determine file type

SYNOPSIS

file [**-c**] [**-f** ffile] [**-m** mfile] arg ...

DESCRIPTION

The *file* command performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable **a.out**, *file* will print the version stamp, provided it is greater than 0.

-c The **-c** option causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under **-c**.

-f If the **-f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

-m The **-m** option instructs *file* to use an alternate magic file.

The *file* command uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of **/etc/magic** explains its format.

Internationalization

The *file* command can classify files containing characters from supplementary code sets. The *file* command reads each argument and can distinguish data files, program text files, shell scripts and executable files as follows:

Files	Classification
Data files containing supplementary characters	data
Shell scripts containing supplementary characters	command text
Language program text files containing literals or comments using supplementary characters	*xxx* text
Executable files	executable

FILES

/etc/magic

SEE ALSO

filehdr(4) in the Programmer's Reference Manual

NAME

find - find files

SYNOPSIS

find path-name-list expression

DESCRIPTION

The *find* command recursively descends the directory hierarchy for each pathname in the *path-name-list* (that is, 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*. Valid expressions are:

-name *file* — True if *file* matches the current filename. 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, only the bits that are set in *onum* are compared with the file permission flags, and the expression evaluates true if they match.

-type *c* — True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, fifo (a.k.a named pipe), or plain file, respectively.

-links *n* — True if the file has *n* links.

-user *uname* — True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the **/etc/passwd** file, it is taken as a user ID.

-group *gname* — True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the **/etc/group** file, it is taken as a group ID.

-size *n*[**c**] — True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a **c**, the size is in characters.

-atime *n* — True if the file has been accessed in *n* days. The access time of directories in *path-name-list* is changed by *find* itself.

-mtime *n* — True if the file has been modified in *n* days.

-ctime *n*	True if the file has been changed in *n* days.
-exec *cmd*	True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument {} is replaced by the current pathname.
-ok *cmd*	Like **-exec** except that the generated command line is printed with a question mark first and is executed only if the user responds by typing **y**.
-print	Always true; causes the current pathname to be printed.
-cpio *device*	Always true; writes the current file on *device* in *cpio* (1) format (5120-byte records).
-newer *file*	True if the current file has been modified more recently than the argument *file*.
-inum n	True if the current file is inode number n.
-depth	Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*(1) to transfer files that are contained in directories without write permission.
-mount	Always true; restricts the search to the file system containing the directory specified, or if no directory was specified, the current directory.
-local	True if the file physically resides on the local system.
(*expression*)	True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

(1) The negation of a primary (! is the unary not operator).

(2) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).

(3) Alternation of primaries (**-o** is the *or* operator).

Internationalization

The *find* command can process characters from supplementary code sets in addition to ASCII characters. Searches are performed on characters, not

individual bytes.

Characters from supplementary code sets can be used in *path-name-list*.

Expressions:

-name *file*
Character from supplementary code sets can be used in *file*.

-exec *cmd*

-ok *cmd*
Characters from supplementary code sets can be used in *cmd*.

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

chmod(1), cpio(1), sh(1), test(1)
stat(2), umask(2), fs(4) in the Programmer's Reference Manual

BUGS

find / -depth always fails with the message:
`find: stat failed: : No such file or directory`

[This page left blank.]

NAME

finger - user information lookup program

SYNOPSIS

finger [options] [name ...]

DESCRIPTION

The *finger* program displays information about users on the local and remote machines. If you don't specify any arguments, *finger* lists the login name, full name (as specified in the fifth field of **/etc/passwd**), 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 logged-in user on the local system. Idle time is in minutes if it is a single integer, hours and minutes if a ':' is present, or days and hours if a 'd' is present.

The *finger* program generates a longer, more detailed format of user information if at least one of the following arguments is supplied:

(1) A user name or a list of user names. A user name can be a login name (first field of **/etc/passwd**) or a first or last name (fifth field of **/etc/passwd**).

(2) The -l option.

The longer format is multi-line, and includes all the information described above, as well as the user's home directory and login shell, any plan specified in the user's **$HOME .plan** file, and any project specified in the user's **$HOME .project** file.

To use *finger* to look up users on a remote machine, specify the user as *user @ host*. A list of users on the same remote host can be enclosed in double quotes; for example,

```
finger "user1 user2 user3"@host
```

If no user names are supplied (argument to *finger* is *@host*); standard, rather than long, format listing is provided.

The following options are recognized by *finger :*

-m Match arguments only on login name.

-l Force long (rather than standard) output format.

-p Suppress printing of the **.plan** files.

-s Force standard (rather than long) output format.

FILES

/etc/utmp	who file
/etc/passwd	for user's names, offices, and so forth
/usr/adm/lastlog	last login times
$HOME/.plan	plans
$HOME/.project	projects

SEE ALSO

who(1)

BUGS

Only the first line of the **.project** file is printed.

There is no way to pass arguments to the remote machine as *finger* uses an internet standard port.

NAME

gencat - generate a formatted message catalog

SYNOPSIS

gencat [-s] [-c] *catfile msgfile...*

DESCRIPTION

The *gencat* utility merges the message text source file(s) *msgfile* into a formatted message catalog *catfile*. The *catfile* will be created if it does not already exist. If *catfile* does exist, its messages will be included in the new *catfile*. If set and message numbers collide, the new message-text defined in *msgfile* will replace the old message text currently contained in *catfile*.

[This page left blank.]

NAME

getopt - parse command options

SYNOPSIS

set -- `getopt** optstring **$*`

DESCRIPTION

WARNING: Start using the new command *getopts* (1) in place of *getopt* (1). The command *getopt* (1) will not be supported in the next major release. For more information, see the **WARNINGS** section which follows.

The *getopt* command is used to break up options in command lines for easy parsing by shell procedures and to check for legal options. The *optstring* is a string of recognized option letters [see *getopt*(3C)]; if a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. The special option -- is used to delimit the end of the options. If it is used explicitly, *getopt* will recognize it; otherwise, *getopt* will generate it; in either case, *getopt* will place it at the end of the options. The positional parameters ($1 $2 ...) of the shell are reset so that each option is preceded by a - and is in its own positional parameter; each option argument is also parsed into its own positional parameter.

Internationalization

Characters from supplementary code sets can be read as the argument to *optstring*. Note, however, that the recognized string of option letters specified in *optstring* must be a single byte character.

EXAMPLES

The following code fragment shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an argument:

```
set -- `getopt abo: $*`
if [ $? != 0 ]
then
        echo $USAGE
        exit 2
fi
for i in $*
do
        case $i in
        -a | -b)        FLAG=$i; shift;;
        -o)             OARG=$2; shift 2;;
```

```
        --)             shift; break;;
        esac
done
```

This code will accept any of the following as equivalent:

```
cmd -aoarg file file
cmd -a -o arg file file
cmd -oarg -a file file
cmd -a -oarg -- file file
```

SEE ALSO

getopts(1), sh(1)
getopt(3C) in the Programmer's Reference Manual

DIAGNOSTICS

The *getopt* command prints an error message on the standard error when it encounters an option letter not included in *optstring*.

WARNINGS

The *getopt*(1) command does not support the part of Rule 8 of the Command Syntax Rules that permits groups of option-arguments following an option to be separated by white space and quoted (see the "About This Document" section at the beginning of this reference manual). For example:

```
cmd -a -b -o "xxx z yy" file
```

is not handled correctly. To correct this deficiency, use the new command *getopts*(1) in place of *getopt*(1).

The *getopt*(1) command will not be supported in the next major release. For this release, a conversion tool has been provided, *getoptcvt*. For more information about *getopts* and *getoptcvt*, see the *getopts*(1) manual page.

If an option that takes an option-argument is followed by a value that is the same as one of the options listed in *optstring* (referring to the earlier EXAMPLES section, but using the following command line: **cmd -o -a file**), *getopt* will always treat **-a** as an option-argument to **-o**; it will never recognize **-a** as an option. For this case, the **for** loop in the example will shift past the *file* argument.

NAME

getopts, getoptcvt - parse command options

SYNOPSIS

getopts optstring name [arg ...]

/usr/lib/getoptcvt [**-b**] file

DESCRIPTION

The *getopts* command is used by shell procedures to parse positional parameters and to check for legal options. It supports all applicable Command Syntax Rules (see Rules 3-10 in the intro(1) manual page) and should be used in place of the *getopt*(1) command. (See "WARNINGS" which follow.)

The *optstring* must contain the option letters the command using *getopts* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

Each time it is invoked, *getopts* will place the next option in the shell variable *name* and the index of the next argument to be processed in the shell variable **OPTIND**. Whenever the shell or a shell procedure is invoked, **OPTIND** is initialized to 1.

When an option requires an option-argument, *getopts* places it in the shell variable **OPTARG**.

If an illegal option is encountered, **?** will be placed in *name*.

When the end of options is encountered, *getopts* exits with a non-zero exit status. The special option "**--**" may be used to delimit the end of the options.

By default, *getopts* parses the positional parameters. If extra arguments (*arg* ...) are given on the *getopts* command line, *getopts* will parse them instead.

The */usr/lib/getoptcvt* command reads the shell script in *file*, converts it to use *getopts* (1) instead of *getopt* (1), and writes the results on the standard output.

-b the results of running */usr/lib/getoptcvt* will be portable to earlier releases of the UNIX system. The */usr/lib/getoptcvt* modifies the shell script in *file* so that when the resulting shell script is executed, it determines at run time whether to invoke *getopts*(1) or *getopt*(1).

So all new commands will adhere to Command Syntax Standards described in intro(1) they should use *getopts* (1) or *getopt* (3C) to parse positional parameters and check for options that are legal for that command (see **WARNINGS**).

Internationalization

Characters from supplementary code sets can be read as the argument to *optstring*.

EXAMPLES

The following fragment of a shell program shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an option-argument:

```
while getopts abo: c
do
        case $c in
        a | b)          FLAG=$c;;
        o)              OARG=$OPTARG;;
        \?)             echo $USAGE
                        exit 2;;
        esac
done
shift `expr $OPTIND - 1`
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o "xxx z yy" file
cmd -a -b -o "xxx z yy" -- file
cmd -ab -o xxx,z,yy file
cmd -ab -o "xxx z yy" file
cmd -o xxx,z,yy -b -a file
```

SEE ALSO

intro(1), sh(1)
getopt(3C) in the Programmer's Reference Manual

WARNINGS

Although the following command syntax rule [see intro(1)] relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the preceding **EXAMPLES**, **a** and **b** are options, and the option **o** requires an option-argument:

`cmd -aboxxx file`
(Rule 5 violation: options with option-arguments must not be grouped with other options.)

`cmd -ab -oxxx file`
(Rule 6 violation: there must be white space after an option that takes an

other options.)

`cmd -ab -oxxx file`

(Rule 6 violation: there must be white space after an option that takes an option-argument.)

Changing the value of the shell variable **OPTIND** or parsing different sets of arguments may lead to unexpected results.

DIAGNOSTICS

The *getopts* command prints an error message on the standard error when it encounters an option letter not included in *optstring*.

[This page left blank.]

NAME

gettxt - retrieve a text string from a message data base

SYNOPSIS

gettxt *msgfile:msgnum* [*dflt_msg*CW]

DESCRIPTION

The *gettxt* command retrieves a text string from a message file in the directory **/lib/locale/***locale***/LC_MESSAGES**. The directory *locale* corresponds to the language in which the text strings are written; see *setlocale*(3C).

msgfile Name of the file in the directory **/lib/locale/***locale***/LC_MESSAGES** to retrieve *msgnum* from. The *msgfile* can be up to 14 characters in length, but may not contain either \0 (null) or the ASCII code for / (slash) or : (colon).

msgnum Sequence number of the string to retrieve from *msgfile*. The strings in *msgfile* are numbered sequentially from *1* to *n* where *n* is the number of strings in the file.

dflt_msg Default string to be used on failure to retrieve the message from the file.

The text string to be retrieved is in the file created by the *mkmsgs*(1M) utility and installed in the *local* directory in **/lib/locale/***locale***/LC_MESSAGE**. The user controls which directory is searched by setting the environment variable **LC_MESSAGES**. If **LC_MESSAGES** is not set, the environment variable LANG will be used. If LANG is not set, the language in which the strings are retrieved is U.S. English and the files containing the strings are in **/lib/locale/C/LC_MESSAGES/***.

If *gettxt* fails to retrieve a message in the requested language, it will try to retrieve the same message in U.S. English. If this also fails, the processing depends on the second argument. If the second argument is not supplied on the command line or it is the null string *gettxt* will display the string Message not found! ! in U.S. English. The second argument will be displayed if it is not the null string.

Nongraphic characters can be included in the default message as alphabetic escape sequences.

EXAMPLES

gettxt UX:1
gettxt UX:1 "hello world"

FILES

/lib/locale/C/LC_MESSAGES/*	U.S. English files created by *mkmsgs*(1M)
/lib/locale/*locale*/LC_MESSAGES/*	message files for different languages created by *mkmsgs*(1M)

SEE ALSO

exstr(1) mkmsgs(1M)

NAME

glossary - definitions of common System V terms and symbols

SYNOPSIS

[help] glossary [term]

DESCRIPTION

The System V Help Facility command *glossary* provides definitions of common technical terms and symbols.

Without an argument, *glossary* displays a menu screen listing the terms and symbols that are currently included in *glossary*. A user may choose one of the terms or may exit to the shell by typing **q** (for "quit"). When a term is selected, its definition is retrieved and displayed. By selecting the appropriate menu choice, the list of terms and symbols can be redisplayed.

A term's definition may also be requested directly from shell level (as shown above), causing a definition to be retrieved and the list of terms and symbols not to be displayed. Some of the symbols must be escaped if requested at shell level in order for the facility to understand the symbol. The following is a table which lists the symbols and their escape sequence.

SYMBOL	ESCAPE SEQUENCE
" "	\" \"
' '	\' \'
[ ]	\[\]
` `	\` \`
#	\#
&	\&
*	*
\	\\
¦	\¦

From any screen in the Help Facility, a user may execute a command via the shell [*sh*(1)] by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable **SCROLL** must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your **.profile** file [see *profile*(4)]:

```
export SCROLL ; SCROLL=no
```

If you later decide that scrolling is desired, **SCROLL** must be set to **yes.**

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

SEE ALSO

help(1), locate(1), sh(1), starter(1), usage(1)
helpadm(1M) in the Administrator's Reference Manual
term(5) in the Programmer's Reference Manual

WARNINGS

If the shell variable **TERM** [see *sh*(1)] is not set in the user's **.profile** file, then **TERM** will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

graph - draw a graph

SYNOPSIS

graph [options]

DESCRIPTION

The *graph* command 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 *tplot*(1G) filters.

If the coordinates of a point are followed by a non-numeric 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 (e.g., the Tektronix 4014: 2 = dotted, 3 = dash-dot, 4 = short-dash, 5 = long-dash).
-s	Save screen, do not erase before plotting.
-x [l]	If **l** is present, the x axis is logarithmic. Next 1 (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

graphics(1G), spline(1G), tplot(1G)

BUGS

The *graph* command stores all points internally and drops those for which there are no room.

Segments that run out of bounds are dropped, not windowed.

Logarithmic axes may not be reversed.

rehash

Causes the internal hash table of the directories' contents 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 be necessary only 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 redirection 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 that have other than a single word as value print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index* 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*

Sets the value of the 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* that 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 match the *unalias* pattern.

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 undesirable 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 nonzero, the commands between the *while* and the matching end are evaluated. Use *break* and *continue* 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 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* argument of *name*. Both *name* and its *index* component must already exist.

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

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

Predefined 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 done explicitly by the user.

Variable	Description
argv	Set to the arguments of the shell; from this variable, positional parameters are substituted, i.e., $1 is replaced by $*argv*[1].
cdpath	Gives a list of alternate directories searched to find subdirectories in *cd* 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 nonbuilt-in commands, all expansions occur before echoing. Built-in 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 that has been referenced in this many events will not be discarded. A *history* that is too large may run the shell out of memory. The last executed command is always saved on the history list.
home	The home directory of the user, initialized from the environment. The filename expansion of ~ refers to this variable.
ignoreeof	If set, the shell ignores the end-of-file from input devices that are terminals. This prevents a shell from accidentally being terminated by typing CTRL-D.
mail	The files where the shell checks for mail. This is done after each command completion results in a prompt, if a specified interval has elapsed. The shell sends the message "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 sends the message "New mail in *name*" when there is mail in the file *name*.

noclobber 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 that are not dealing with filenames or after a list of filenames has been obtained and further expansions are not desirable.

nomatch 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 pathnames will execute. The usual search path is **/bin**, **/usr/bin**, and ., but this may vary from system to system. For the superuser, the default search path is **/etc**, **/bin** and **/usr/bin**. A shell that is given neither the **-c** nor the **-t** option will normally hash the contents of the directories in the *path* variable after reading **.cshrc** and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash*, or the commands may not be found.

prompt The string that is printed before each command is read from an interactive terminal input. If a ! appears in the string it will be replaced by the current event number unless a preceding \ is given. The default is % or # for the superuser.

shell The file in which the shell resides. This is used in forking shells to interpret files that have execute bits set but are not executable by the system. (See the section "Nonbuilt-In Command Execution" in this manpage.) The *shell* is 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. Abnormal

termination results in a core dump. Built-in commands that fail return exit status 1; all other built-in commands set status 0.

time Controls automatic timing of commands. If set, then any command that takes more than this many CPU seconds will cause a line giving user, system, and real times and a utilization percentage (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.

The shell copies the environment variable PATH into the variable *path* and copies the value back into the environment whenever *path* is set. Thus, it is not necessary to worry about its setting other than in the file **.cshrc** as inferior *csh* processes will import the definition of *path* from the environment.

Nonbuilt-In Command Execution

When a command to be executed is found to not be a built-in command, the shell attempts to execute the command via *exec*(S). 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 pathname 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 *cd* 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 and 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 pathname 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:

Flag	Description
-c	Reads commands from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.
-e	Causes the shell to exit if any invoked command terminates abnormally or yields a nonzero exit status.
-f	Lets the shell start faster because it will neither search for nor execute commands from the file **.cshrc** in the user's home directory.
-i	Makes the shell interactive. The shell prompts for its top-level input even if it appears not to be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.
-n	Causes commands to be parsed but not executed. This may aid in syntactic checking of shell scripts.
-s	Causes command input to be taken from the standard input.
-t	Reads and executes a single line of input. A backslash (\) can 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	Causes the *echo* variable to be set even before **.cshrc** is executed.

After processing of flag arguments, if arguments remain but none of the **-c**, **-i**, **-s**, or **-t** options were 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 on a typical system most shell scripts are written for the standard shell [see *sh*(1)], the C shell executes such a standard shell if the first character of a script is not a #: that is, 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 that the shell inherited from its parent. The shell's 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**.

NEW ENVIRONMENT VARIABLES

The new environment variable described in this section has been added to the C shell. The C shell will behave normally for those users who do not set **DOSPATH**. Users who want to execute DOS programs directly from the C shell, that is, bypassing the normal DOS bootup that occurs when running *vpix*, should set **DOSPATH** to include those directories in **PATH** that contain DOS executables.

DOSPATH is a string with the same format as **PATH**; it contains a subset of the list of directories from **PATH**. When searching a directory in **PATH** for a program, the C shell determines whether that directory is also in **DOSPATH**. If it is not, the C shell acts as usual. If it is, the C shell looks first for the command with the suffix **.com**, then **.exe**, then **.bat**, and finally, for the command without any suffix. Whenever the result of a path search gives a file with one of these DOS suffixes, the shell runs the *vpix* program via a standard search path and adds arguments -c and the full pathname of the DOS program (including the suffix).

For example, if **PATH** is set to **:/bin:/usr/bin**, **DOSPATH** is set to ., the current directory is **/usr/john/dosbin**, and there is a DOS program named *abc.comi* in the current directory, then typing *abc* to the C shell will cause the command *vpix -c /usr/john/dosbin/abc.com* to be executed, which will run the DOS program *abc.com* without the normal *vpix* DOS bootup.

FILES

~/.cshrc	Read by each shell at the beginning of execution
~/.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
/etc/cshrc	Default file of automatically invoked commands

NOTES

Words can be no longer than 512 characters. The number of arguments to a command which involves filename expansion is limited to 1/6 number of characters allowed in an argument list, which is 5120 less the characters in the environment. 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* substitutions on a single line to 20.

Built-in control structure commands like *foreach* and *while* cannot be used with the pipe symbol (|), ampersand (&), or semicolon (;).

Commands within loops prompted for by ? are not placed in the *history* list.

It is not possible to use the colon (:) modifiers on the output of command substitutions.

The *csh* interpreter attempts to import and export the PATH variable for use with regular shell scripts. This only works for simple cases, where the PATH contains no command characters.

This version of *csh* does not support or use the process control features of the 4th Berkeley Distribution.

You can modify the list of commands that *csh* automatically invokes by editing the **/etc/default/.cshrc** file. For example, if you want to automatically assign the **alias** *h* to the **history** command, add the following line to the **/etc/default/.cshrc** file using the computer editor of your choice:

```
alias history h
```

SEE ALSO

umask(1), wait(1)
access(2), exec(2), fork(2), pipe(2), ssignal(3C), a.out(4), environ(5) in the Programmer's Reference Manual

[This page left blank.]

NAME

csplit - context split

SYNOPSIS

csplit [-s] [-k] [-f prefix] file arg1 [... argn]

DESCRIPTION

The *csplit* command reads *file* and separates it into n+1 sections, defined by the arguments *arg1*... *argn*. By default the sections are placed in xx00 ... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.

:

:

n+1: From the line referenced by *argn* to the end of *file*.

If the *file* argument is a -, then standard input is used.

The options to *csplit* are:

-s The *csplit* command normally prints the character counts for each file created. If the -s option is present, *csplit* suppresses the printing of all character counts.

-k The *csplit* command normally removes created files if an error occurs. If the **-k** option is present, *csplit* leaves previously created files intact.

-f *prefix* If the **-f** option is used, the created files are named *prefix*00 ... *prefix*n. The default is xx00 ... xx*n*.

The arguments (*arg1* ... *argn*) to *csplit* can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or - some number of lines (e.g., **/Page/-5**).

%rexp% This argument is the same as */rexp/*, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*} Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded newlines. The *csplit* command does not affect the original file; it is the user's responsibility to remove it.

EXAMPLES

```
csplit -f cobol file '/procedure division/' /par5./ /par16./
```

The preceding example creates four files, **cobol00 ... cobol03**. After editing the split files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that the preceding example overwrites the original file.

The next example would split the file at every 100 lines, up to 10,000 lines. The **-k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed:

```
csplit -k file 100 {99}
```

Assuming that **prog.c** follows the normal C coding convention of ending routines with a } at the beginning of the line, the following example creates a file containing each separate C routine (up to 21) in **prog.c**:

```
csplit -k prog.c '%main(%' '/^}/+1' {20}
```

SEE ALSO

ed(1), sh(1)
regexp(5) in the Programmer's Reference Manual

DIAGNOSTICS

Self-explanatory except for:

arg - out of range

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

ct - spawn getty to a remote terminal

SYNOPSIS

ct [**-wn**] [**-xn**] [**-h**] [**-v**] [-sspeed] telno ...

DESCRIPTION

The *ct* command dials the telephone number of a modem that is attached to a terminal, and spawns a *getty* process to that terminal. The *telno* command is a telephone number, with equal signs for secondary dial tones and minus signs for delays at appropriate places. (The set of legal characters for *telno* is 0 through 9, -, =, *, and #. The maximum length *telno* is 31 characters). If more than one telephone number is specified, the *ct* command tries each in succession until one answers; this is useful for specifying alternate dialing paths.

The *ct* command will try each line listed in the file **/usr/lib/uucp/Devices** until it finds an available line with appropriate attributes or runs out of entries. If there are no free lines, *ct* asks if it should wait for one, and if so, for how many minutes it should wait before it gives up. The *ct* command continues to try to open the dialers at one-minute intervals until the specified limit is exceeded. The dialogue may be overridden by specifying the *-wn* option, where *n* is the maximum number of minutes that *ct* is to wait for a line.

The *-xn* option is used for debugging; it produces a detailed output of the program execution on stderr (standard error). The debugging level, *n*, is a single digit; **-x9** is the most useful value.

Normally, *ct* hangs up the current line, so the line can answer the incoming call. The **-h** option prevents this action. The **-h** option also waits for the termination of the specified *ct* process before returning control to the user's terminal. If the **-v** option is used, *ct* sends a running narrative to the standard error output stream.

The data rate may be set with the -s option, where *speed* is expressed in baud. The default rate is 1200.

After the user on the destination terminal logs out, there are two things that could occur depending on what type of getty is on the line (*getty* or *uugetty*). For the first case, *ct* prompts, `Reconnect?` If the response begins with the letter **n**, the line is dropped; otherwise, *getty* is started again and the **login:** prompt is printed. In the second case, there is already a getty (*uugetty*) on the line, so the **login:** message appears.

To log out properly, the user must type <Ctrl> d.

Of course, the destination terminal must be attached to a modem that can answer the telephone.

FILES

/usr/lib/uucp/Devices
/usr/adm/ctlog

SEE ALSO

cu(1C), login(1), uucp(1C)
getty(1M), uugetty(1M) in the Administrator's Reference Manual

BUGS

For a shared port, one used for both dial-in and dial-out, the *uugetty* program running on the line must have the **-r** option specified [see *uugetty*(1M)].

NAME

cu - call another UNIX system

SYNOPSIS

cu [**-s**speed] [**-l**line] [**-h**] [**-t**] [**-d**] [**-o** | **-e**] [**-n**] telno
cu [**-s** speed] [**-h**] [**-d**] [**-o** | **-e**] **-l** line
cu [**-h**] [**-d**] [**-o** | **-e**] systemname

DESCRIPTION

The *cu* command calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of ASCII files.

The *cu* command accepts the following options and arguments:

-s*speed* Specifies the transmission speed (300, 1200, 2400, 4800, 9600); the default value is "Any" speed which will depend on the order of the lines in the **/usr/lib/uucp/Devices** file. Most modems are either 300 or 1200 baud. Directly connected lines may be set to a speed higher than 1200 baud.

-l*line* Specifies a device name to use as the communication line. This can be used to override the search that would otherwise take place for the first available line having the right speed. When the **-l** option is used without the **-s** option, the speed of a line is taken from the **Devices** file. When the **-l** and **-s** options are both used together, *cu* will search the **Devices** file to check if the requested speed for the requested line is available. If so, the connection is made at the requested speed; otherwise, an error message is printed and the call is not made. The specified device is generally a directly connected asynchronous line (e.g., **/dev/tty***ab*) in which case a telephone number (*telno*) is not required. The specified device need not be in the **/dev** directory. If the specified device is associated with an auto dialer, a telephone number must be provided. Use of this option with *systemname* rather than *telno* does not give the desired result (see *systemname* in following text).

-h Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

-t Used to dial an ASCII terminal which has been set to auto answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.

-d Causes diagnostic traces to be printed.

-o Designates that odd parity is to be generated for data sent to the remote system.

-e Designates that even parity is to be generated for data sent to the remote system.

-n For added security, it prompts the user to provide the telephone number to be dialed rather than taking it from the command line.

telno When using an automatic dialer, the argument is the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of four seconds.

systemname A *uucp* system name may be used rather than a telephone number; in this case, *cu* will obtain an appropriate direct line or telephone number from **/usr/lib/uucp/Systems.** Note: The *systemname* option should not be used in conjunction with the **-l** and **-s** options as *cu* will connect to the first available line for the system name specified, ignoring the requested line and speed.

After making the connection, *cu* runs as two processes:

The *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system.

The *receive* process accepts data from the remote system and, except for lines beginning with ~, passes it to the standard output.

Normally, an automatic DC3/DC1 protocol is used to control input from the remote system so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following user-initiated commands:

~. terminate the conversation.

~! escape to an interactive shell on the local system.

~!*cmd...* run *cmd* on the local system (by **sh -c**).

~$*cmd...* run *cmd* locally and send its output to the remote system.

~%cd change the directory on the local system. Note: The **~!cd** will cause the command to be run by a sub-shell, probably not what was intended.

~ **%take** *from* [*to*]	copy file *from* (on the remote system) to file *to* on the local system. If *to* is omitted, the *from* argument is used in both places.
~ **%put** *from* [*to*]	copy file *from* (on local system) to file *to* on remote system. If *to* is omitted, the *from* argument is used in both places. For both ~ **%take** and **put** commands, as each block of the file is transferred, consecutive single digits are printed to the terminal.
~ ~ *line*	send the line ~ *line* to the remote system.
~ **%break**	transmit a **BREAK** to the remote system (which can also be specified as ~ **%b**).
~ **%debug**	toggles the **-d** debugging option on or off (which can also be specified as ~ **%d**).
~ **t**	prints the values of the termio structure variables for the user's terminal (useful for debugging).
~ **l**	prints the values of the termio structure variables for the remote communication line (useful for debugging).
~ **%nostop**	toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one which does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output. Internally the program accomplishes this by initiating an output diversion to a file when a line from the remote begins with ~. The complete sequence is:

~ > [>]: *file*
zero or more lines to be written to *file*
~ >

Data from the remote is diverted (or appended, if > > is used) to *file* on the local system. The trailing ~ > marks the end of the diversion.

The use of ~ **%put** requires *stty*(1) and *cat*(1) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current control characters on the local system. Backslashes are inserted at appropriate places.

The use of ~%take requires the existence of *echo*(1) and *cat*(1) on the remote system. Also, tabs mode [see *stty*(1)] should be set on the remote system if tabs are to be copied without expansion to spaces.

When *cu* is used on system X to connect to system Y and subsequently used on system Y to connect to system Z, commands on system Y can be executed by using ~ ~. Executing a tilde command reminds the user of the local system uname. For example, *uname* can be executed on Z, X, and Y as follows:

```
uname
Z
~[X]!uname
X
~ ~[Y]!uname
Y
```

In general, ~ causes the command to be executed on the original machine, ~ ~ causes the command to be executed on the next machine in the chain.

EXAMPLES

To dial a system whose telephone number is 9 201 555 1212 using 1200 baud (where a dial tone is expected after the 9):

```
cu -s1200 9=12015551212
```

If the speed is not specified, "Any" is the default value.

To log in to a system connected by a direct line, enter:

```
cu -l /dev/ttyXX
```

or

```
cu -l ttyXX
```

To dial a system with the specific line and a specific speed, enter:

```
cu -s1200 -l ttyXX
```

To dial a system using a specific line associated with an auto dialer, enter:

```
cu -l culXX 9=12015551212
```

To use a system name, enter:

```
cu systemname
```

FILES

/usr/lib/uucp/Systems
/usr/lib/uucp/Devices
/usr/spool/locks/LCK..(tty-device)

SEE ALSO

cat(1), ct(1C), echo(1), stty(1), uucp(1C), uname(1)

DIAGNOSTICS

Exit code is zero for normal exit, otherwise, one.

WARNINGS

The *cu* command buffers input data internally and does not do any integrity checking on data it transfers. Data fields with special *cu* characters may not be transmitted properly. Depending on the interconnection hardware, it may be necessary to use a ~. to terminate the conversion even if **stty 0** has been used. Non-printing characters are not dependably transmitted using either the **~%put** or **~%take** commands. The *cu* command between some modems will not return a login prompt immediately upon connection. A carriage return will return the prompt.

BUGS

During the **~%put** operation, there is an artificial slowing of transmission by *cu* so that loss of data is unlikely.

[This page left blank.]

NAME

cut - cut out selected fields of each line of a file

SYNOPSIS

cut -clist [file ...]
cut -flist [**-d** char] [**-s**] [file ...]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (**-c** option), or the length can vary from line to line and be marked with a field delimiter character like *tab* (**-f** option). The *cut* command can be used as a filter; if no files are given, the standard input is used. In addition, a filename of **-** explicitly refers to standard input.

The meanings of the options are:

list A comma-separated list of integer field numbers (in increasing order), with optional **-** to indicate ranges [e.g., **1,4,7**; **1-3,8**; **-5,10** (short for **1-5,10**); or **3-** (short for third through last field)].

-c*list* The *list* following **-c** (no space) specifies character positions (e.g., **-c1-72** would pass the first 72 characters of each line).

-f*list* The *list* following **-f** is a list of fields assumed to be separated in the file by a delimiter character (see **-d**); e.g., **-f1,7** copies the first and seventh field only. Lines with no field delimiters are passed through intact (useful for table subheadings), unless **-s** is specified.

-d*char* The character following **-d** is the field delimiter (**-f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.

-s Suppresses lines with no delimiter characters in case of **-f** option. Unless specified, lines with no delimiters are passed through untouched.

Either the **-c** or **-f** option must be specified.

Use *grep*(1) to make horizontal cuts (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut -d: -f1,5 /etc/passwd	mapping of user IDs to names
name= `who am i | cut -f1 -d" " `	to set **name** to current login name.

SEE ALSO

grep(1), paste(1)

DIAGNOSTICS

ERROR: line too long
A line can have no more than 1023 characters or fields, or there is no newline character.

ERROR: bad list for c/f option
Missing **-c** or **-f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

ERROR: no fields The *list* is empty.

ERROR: no delimiter
Missing *char* on **-d** option.

ERROR: cannot handle multiple adjacent backspaces
Adjacent backspaces cannot be processed correctly.

WARNING: cannot open <filename>
Either *filename* cannot be read or does not exist. If multiple filenames are present, processing continues.

NAME

cw, checkcw – prepare constant-width text for troff

SYNOPSIS

cw [**-lxx**] [**-rxx**] [**-fn**] [**-t**] [**+t**] [**-d**] [files]

checkcw [**-lxx**] [**-rxx**] files

DESCRIPTION

The *cw* utility is a preprocessor for *troff* input files that contain text to be typeset in the constant-width (CW) font. Note that the *cw* utility has been superseded and is no longer required in the current version of Documenter's Workbench (2.0).

Text typeset with the CW font resembles the output of terminals and of line printers. This font is used to typeset examples of programs and computer output in user manuals, programming texts, and so forth. It has been designed to be quite distinctive (but not overly obtrusive) when used together with the Times Roman font.

Because the CW font contains a "non-standard" set of characters and because text typeset with it requires different character and inter-word spacing than is used for "standard" fonts, documents that use the CW font must be preprocessed by *cw*.

The CW font contains the 94 printing ASCII characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! $ %&( ) ‘ ’ * + @. , / : ; = ? [ ] | - _ ^ ~ " < > { } # \
```

plus eight non-ASCII characters represented by four-character *troff* names (in some cases attaching these names to "non-standard" graphics):

Character	Symbol	Troff Name
"Cents" sign	¢	\(ct
EBCDIC "not" sign	¬	\(no
Left arrow	←	\(<-
Right arrow	→	\(->
Down arrow	↓	\(da
Vertical single quote	’	\(fm
Control-shift indicator	†	\(dg
Visible space indicator	□	\(sq
Hyphen	-	\(hy

The hyphen is a synonym for the unadorned minus sign (-). Certain versions of *cw* recognize two additional names: **\(ua** for an up arrow (↑) and **\(lh** for a diagonal left-up (home) arrow.

The *cw* utility recognizes five request lines, as well as user-defined delimiters. The request lines look like *troff* macro requests, and are copied in their entirety by *cw* onto its output; thus, they can be defined by the user as *troff* macros; in fact, the .CW and .CN macros *should* be so defined (see "HINTS" in this manpage). The five requests are:

.CW Start of text to be set in the CW font; **.CW** causes a break; it can take precisely the same options, in precisely the same format, as are available on the *cw* command line.

.CN End of text to be set in the CW font; **.CN** causes a break; it can take the same options as are available on the *cw* command line.

.CD Change delimiters and/or settings of other options; takes the same options as are available on the *cw* command line.

.CP *arg1 arg2 arg3 ... argn*
All the arguments (which are delimited like *troff* macro arguments) are concatenated, with the odd-numbered arguments set in the CW font and the even-numbered ones in the prevailing font.

.PC *arg1 arg2 arg3 ... argn*
Same as **.CP**, except that the even-numbered arguments are set in the CW font and the odd-numbered ones in the prevailing font.

The .CW and .CN requests are meant to bracket text (for example, a program fragment) that is to be typeset in the CW font "as is." Normally, *cw* operates in the *transparent* mode. In that mode, except for the .CD request and the nine special four-character names listed in the table above, every character between .CW and .CN request lines stands for itself. In particular, *cw* arranges for periods (.) and apostrophes (') at the beginning of lines, and backslashes (\) everywhere to be "hidden" from *troff*. The transparent mode can be turned off (see below), in which case normal *troff* rules apply; in particular, lines that begin with . and ' are passed through untouched (except if they contain delimiters–see below). In either case, *cw* hides the effect of the font changes generated by the .CW and .CN requests; *cw* also defeats all ligatures (fi, ff, and so forth) in the CW font.

The only purpose of the .CD request is to allow the changing of various options other than just at the beginning of a document.

The user can also define *delimiters*. The left and right delimiters perform the same function as the .CW/.CN requests; they are meant, however, to enclose CW "words" or "phrases" in running text (see example under *BUGS* below). The *cw* preprocessor treats text between delimiters in the same manner as text enclosed by .CW/.CN pairs, except that, for aesthetic reasons, spaces and backspaces inside .CW/.CN pairs have the same width as other CW characters, while spaces and backspaces between delimiters are half as wide, so they have the same width as spaces in the prevailing text (but are *not* adjustable). Font changes due to delimiters are *not* hidden.

Delimiters have no special meaning inside .CW/.CN pairs.

The options are:

–l*xx* The one- or two-character string *xx* becomes the left delimiter; if *xx* is omitted, the left delimiter becomes undefined.

–r*xx* Same for the right delimiter. The left and right delimiters may (but need not) be different.

–f*n* The CW font is mounted in font position *n*; acceptable values for *n* are 1, 2, and 3 (default is 3, replacing the bold font). This option is only useful at the beginning of a document.

–t Turn transparent mode *off*.

+t Turn transparent mode *on* (this is the initial default).

–d Print current option settings on file descriptor 2 in the form of *troff* comment lines. This option is meant for debugging.

The *cw* preprocessor reads the standard input when no *files* are specified (or when – is specified as the last argument), so it can be used as a filter. Typical usage is:

cw *files* | troff ...

The *checkcw* utility checks that left and right delimiters, as well as the .CW/.CN pairs, are properly balanced. It prints out all offending lines.

HINTS

Typical definitions of the .CW and .CN macros meant to be used with the *mm*(5) macro package:

```
.de CW
.DS I
.ps 9
.vs 10.5p
.ta 16m/3u 32m/3u 48m/3u 64m/3u 80m/3u 96m/3u ...
..
.de CN
.ta .5i 1i 1.5i 2i 2.5i 3i ...
.vs
.ps
.DE
..
```

At the very least, the .CW macro should invoke the *troff* no-fill (**.nf**) mode.

When set in running text, the CW font is meant to be set in the same point size as the rest of the text. In displayed matter, on the other hand, it can often be profitably set one point *smaller* than the prevailing point size (the displayed definitions of .CW and .CN above are one point smaller than the running text on this page). The CW font is sized so that, when it is set in 9-point, there are 12 characters per inch.

Documents that contain CW text may also contain tables and/or equations. If this is the case, the order of preprocessing should be: *cw*, *tbl*, and *eqn*. Usually, the tables contained in such documents will not contain any CW text, although it is entirely possible to have *elements* of the table set in the CW font; of course, care must be taken that *tbl*(1) format information not be modified by *cw*. Attempts to set equations in the CW font are not likely to be either pleasing or successful.

In the CW font, overstriking is most easily accomplished with backspaces: letting the <left arrow> key represent a backspace, striking d, then the <left arrow> key twice, and then the <Ctrl>-<Shift> combination yields an overstrike over d. [Because backspaces are half as wide between delimiters as inside .CW/.CN pairs (see previous text), two backspaces are required for each overstrike between delimiters.]

FILES

/usr/lib/font/ftCW CW font-width table

SEE ALSO

Documenter's Workbench User's Guide
Documenter's Workbench Technical Discussion and Reference

WARNINGS

If text preprocessed by *cw* is to make any sense, it must be set on a typesetter equipped with the CW font or on a STARE facility; on the latter, the CW font appears as bold, but with the proper CW spacing.

BUGS

It is not a good idea to use periods (.), backslashes (\), or double quotes (") as delimiters, or as arguments to .CP and .PC.

Certain CW characters do not concatenate gracefully with certain Times Roman characters, for example, a CW ampersand (&) followed by a Times Roman comma (,); in such cases, judicious use of *troff* half- and quarter-spaces (\| and \^) is most salutary; for example, one should use _&_\^, (rather than just plain _&_,) to obtain &, (assuming that _ is used for both delimiters).

The *cw* utility is not compatible with *nroff*.

The output of *cw* is hard to read.

[This page left blank.]

NAME

date - print and set the date

SYNOPSIS

date [+ format]
date [mmddhhmm[[yy] | [ccyy]]]

DESCRIPTION

If no argument is given, or if the argument begins with +, the current date and time are printed. Otherwise, the current date is set (only by the superuser). 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; *cc* is the century minus one and is optional; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. The *date* command takes care of the conversion to and from local standard and daylight saving time. Only the superuser may change the date.

If the argument begins with +, the output of *date* is under the control of the user. All output fields are of fixed size (zero-padded, if necessary). Each field descriptor is preceded by % and is replaced in the output by its corresponding value. A single % is encoded by %%. All other characters are copied to the output without change. The string is always terminated with a newline character. If the argument contains embedded blanks, it must be quoted (see "EXAMPLES" in this manpage).

Specifications of native language translations of month and weekday names are supported. The language used depends on the value of the environment variable **LANGUAGE** [see *environ*(5)]. The month and weekday names used for a language are taken from strings in the file for that language in the **/lib/cftime** directory [see *cftime*(4)].

After successfully setting the date and time, *date* displays the new date according to the format defined in the environment variable **CFTIME** [see *environ*(5)].

Field descriptors (must be preceded by a %):

- **a** abbreviated weekday name
- **A** full weekday name
- **b** abbreviated month name

B	full month name
d	day of month - 01 to 31
D	date as mm/dd/yy
e	day of month - 1 to 31 (single digits are preceded by a blank)
h	abbreviated month name (alias for %b)
H	hour - 00 to 23
I	hour - 01 to 12
j	day of year - 001 to 366
m	month of year - 01 to 12
M	minute - 00 to 59
n	insert a newline character
p	string containing ante-meridiem or post-meridiem indicator (by default, AM or PM)
r	time as *hh:mm:ss pp* where *pp* is the ante-meridiem or post-meridiem indicator (by default, AM or PM)
R	time as hh:mm
S	second - 00 to 59
t	insert a tab character
T	time as *hh:mm:ss*
U	week number of year (Sunday as the first day of the week) - 01 to 52
w	day of week - Sunday = 0
W	week number of year (Monday as the first day of the week) - 01 to 52
x	Country-specific date format
X	Country-specific time format
y	year within century - 00 to 99
Y	year as *ccyy* (4 digits)
Z	timezone name

EXAMPLE

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

would have generated as output:

```
DATE: 08/01/76
TIME: 14:45:05
```

FILES

/dev/kmem

NOTE

Administrators should note the following; if you attempt to set the current date to one of the dates that the standard and alternate time zones change (for example, the date that daylight time is starting or ending), and you attempt to set

the time to a time in the interval between the end of standard time and the beginning of the alternate time (or the end of the alternate time and the beginning of standard time), the results are unpredictable.

SEE ALSO

cftime(4), environ(5) in the Programmer's Reference Manual

DIAGNOSTICS

No permission	if you are not the superuser and you try to change the date
bad conversion	if the date set is syntactically incorrect
bad format character	if the field descriptor is not recognizable

[This page left blank.]

NAME

dc - desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

The *dc* command 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. [See *bc*(1), a preprocessor for *dc* that provides infix notation and a C-like syntax that implements functions. The *bc* command also provides reasonable control structures for programs.] 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 are printed.

q
: Exits the program. If executing a string, the recursion level is popped by two.

Q Exits the program. 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 evaluated 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 UNIX system 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 is reasonable if all bases 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.

; : Used by *bc*(1) for array operations.

EXAMPLE

This example prints the first ten values of n!:

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc(1), hc(1)

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.

[This page left blank.]

NAME

deroff - remove nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [**-mx**] [**-w**] [files]

DESCRIPTION

The *deroff* command reads each of the *files* in sequence and removes all *troff*(1) requests, macro calls, backslash constructs, *eqn*(1) constructs (between **.EQ** and **.EN** lines and between delimiters), and *tbl*(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output.

The *deroff* command follows chains of included files (**.so** and **.nx** *troff* commands); if a file has already been included, a **.so** naming that file is ignored and a **.nx** naming that file terminates execution. If no input file is given, *deroff* reads the standard input.

The **-m** option may be followed by an **m**, **s**, or **l**. The **-mm** option causes the macros to be interpreted so that only running text is output (i.e., no text from macro lines). The **-ml** option forces the **-mm** option and also causes deletion of lists associated with the **mm** macros.

If the **-w** option is given, the output is a word list, one word per line, with all other characters deleted. Otherwise, the output follows the original with the deletions mentioned above. In text, a word is any string that contains at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a word is a string that begins with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from words.

BUGS

The *deroff* command is not a complete *troff* interpreter, so it can be confused by subtle constructs. Most of these errors result in too much rather than too little output.

The **-ml** option does not handle nested lists correctly.

[This page left blank.]

NAME

diff - differential file comparator

SYNOPSIS

diff [**-efbh**] file1 file2

DESCRIPTION

The *diff* command 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 with the name *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 comes 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 recreates *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

bdiff(1), cmp(1), comm(1), ed(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some differences, 2 for trouble.

WARNINGS

Missing newline at end of file X

Indicates that the last line of file X did not have a newline. If the lines are different, they are flagged and output although the output seems to indicate they are the same.

BUGS

Editing scripts produced under the **-e** or **-f** option are naive about creating lines consisting of a single period (.).

NAME

diff3 - 3-way differential file comparison

SYNOPSIS

diff3 [**-ex3**] file1 file2 file3

DESCRIPTION

The *diff3* command compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* 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 file is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

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 incorporates into *file1* 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 *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

FILES

/tmp/d3*
/usr/lib/diff3prog

SEE ALSO

diff(1)

BUGS

Text lines that consist of a single . defeat **-e**.
Files longer than 64K bytes do not work.

NAME

greek - select terminal filter

SYNOPSIS

greek [-T*terminal*]

DESCRIPTION

The *greek* command is a filter that reinterprets the extended character set, as well as the reverse and half-line motions, of a 128-character TELETYPE Model 37 terminal for certain other terminals. Special characters are simulated by overstriking, if necessary and possible. If the argument is omitted, *greek* attempts to use the environment variable **$TERM** [see *environ*(5)]. Currently, the following *terminal*s are recognized:

300	DASI 300
300-12	DASI 300 in 12-pitch
300s	DASI 300s
300s-12	DASI 300s in 12-pitch
450	DASI 450
450-12	DASI 450 in 12-pitch
1620	Diablo 1620 (alias DASI 450)
1620-12	Diablo 1620 (alias DASI 450) in 12-pitch
2621	Hewlett-Packard 2621, 2640, and 2645
2640	Hewlett-Packard 2621, 2640, and 2645
2645	Hewlett-Packard 2621, 2640, and 2645
4014	Tektronix 4014
hp	Hewlett-Packard 2621, 2640, and 2645.
tek	Tektronix 4014.

FILES

/usr/bin/300
/usr/bin/300s
/usr/bin/4014
/usr/bin/450
/usr/bin/hp

SEE ALSO

300(1), 4014(1), 450(1), hp(1), tplot(1G)
environ(5), term(5) in the Programmer's Reference Manual

[This page left blank.]

NAME

grep - search a file for a pattern

SYNOPSIS

grep [options] limited regular expression [file ...]

DESCRIPTION

The *grep* command searches files for a pattern and prints all lines that contain that pattern. The *grep* command uses limited regular expressions (expressions that have string values that use a subset of the possible alphanumeric and special characters) like those used with *ed*(1) to match the patterns. It uses a compact non-deterministic algorithm.

Be careful using the characters $, *, [, ^, |, (,), and \ in the *limited regular expression* because they are also meaningful to the shell. It is safest to enclose the entire *limited regular expression* in single quotes '...'.

If no files are specified, *grep* assumes standard input. Normally, each line found is copied to standard output. The filename is printed before each line found if there is more than one input file.

Command line options are:

- **-b** Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).
- **-c** Print only a count of the lines that contain the pattern.
- **-i** Ignore upper/lowercase distinction during comparisons.
- **-h** Prevents the name of the file containing the matching line from being appended to that line. Used when searching multiple files.
- **-l** Print the names of files with matching lines once, separated by newlines. Does not repeat the names of files when the pattern is found more than once.
- **-n** Precede each line by its line number in the file (first line is 1).
- **-s** Suppress error messages about nonexistent or unreadable files.
- **-v** Print all lines except those that contain the pattern.

Internationalization

The *grep* command can process characters from supplementary code sets, as well as ASCII characters. Searches are performed on characters, not individual bytes.

Option:

- **-i** Ignores upper/lowercase distinction during comparisons, is valid for single-byte characters only.

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 (even if matches were found).

BUGS

Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h**, which is included as part of the basic software development set.

If there is a line with embedded nulls, *grep* will only match up to the first null; if it matches, it will print the entire line.

NAME

gsar - graphical system activity reporter

SYNOPSIS

gsar [-n] [t[n]]
gsar [-n] [t[n]]

DESCRIPTION

The *gsar* reporter samples cumulative activity counters in the operating system and displays the resulting data graphically on Unisys supported terminals. The default display mode is 132 columns. If the **-n** option is selected, the display uses 80 columns. In the 132 column mode, the data is presented dynamically on four logical screens (A, B, C, and D) as bar graphs with moving average indicators (vertical scaled lines) to the right of the bar graphs. In 80 column mode, a fifth screen (E) is added.

To select a logical screen, type 'a', 'b', 'c', 'd', or 'e'. To toggle between screens, press the escape key.

The *gsar* reporter, invoked with the first syntax, samples for *n* intervals of *t* seconds. The default value of *t* is 1. The default value of *n* is 100.

The *gsar* reporter, invoked with the second syntax, samples for *n* intervals of *t* seconds and displays on the logical screen selected by the *-S* option. The default value of *t* is 1. The default value of *n* is 100.

OPTIONS

-Ss Select logical screen, where s is:

0	logical screen A
1	logical screen B
2	logical screen C
3	logical screen D
4	logical screen E (valid only in 80 columns mode)

LOGICAL SCREENS

The exact screen that is displayed depends upon the **-n** option. All screens display certain data with column headers matching display columns as follows:

Master CPU utilization

usr	Portion of time running in user mode.

sys Portion of time running in system mode.

wio Portion of time idle with some process waiting for block I/O.

idle Portion of time otherwise idle.

Tables

prc Portion of process table filled.

ino Portion of inode table filled.

fil Portion of file table filled.

All tables are displayed on a percent basis and display "OV!" at the top of the column if an overflow is detected.

Memory

mem Portion of user memory being used.

Cache

rch Percent of read cache match/system buffers.

wch Percent of write cache match/system buffers.

cls Number of system calls per second.

rds Number of read calls per second.

wts Number of write calls per second.

frk Number of forks per second.

exc Number of execs per second.

rcs Number of characters read per second.

wcs Number of characters written per second.

igt Number of times the kernel function iget was called per second.

nmi Number of times the kernel function namei was called per second.

dbk Number of times the kernel function dirblk was called per second.

rqz Average run queue length while occupied.

sqz Average swap queue length while occupied.

psw Number of process switches per second.

raw Input character rate.

can Input character rate processed by canon.

out Output character rate.

rcv Receive interrupt rates.

xmt Transmit interrupt rates.

mdm Modem interrupt rates.

Slave Processor

S1 User and system time from slave CPU1.

S2 User and system time from slave CPU2.

S3 Uwer and system time from slave CPU3.

The only information displayed for slave processors is usr and sys time. Any remaining time can be considered idle time. If a slave processor is not installed, then "NI" is displayed at the bottom of the respective column.

EXAMPLES

To view system activity for ten minutes, with samples every 5 seconds and the data presented on logical screen C:

```
gsar -S2 5 120
```

To view system activity for 100 seconds, with samples every second and the data presented on logical screen A (default situation):

```
gsar
```

or:

```
sar -S0 1 100
```

or:

```
gsar 1 100
```

FILES

/usr/bin/gsar

/usr/lib/sa/gsadc

SEE ALSO

sar(1M).

RESTRICTIONS

This package supports only Unisys terminals. Terminal environment variables must equal svt1210 or svt1220. Some ANSI look-alikes may work (e.g. Wyse75) when running in ANSI mode, but the terminal environment variables must be as stated above.

Occasionally, a terminal may get stuck in the middle of painting a box. If this happens, use ^Q (control-Q) to free it.

NAME

hd - display files in hexadecimal format

SYNOPSIS

hd [*-format* [**-s** *offset*] [**-n** *count*] [*file*] ...

DESCRIPTION

The *hd* command displays the contents of files in hexadecimal, octal, decimal, and character formats. Control over the specification of ranges of characters is also available. The default behavior is with the following flags set: **-abx -A**. This says that addresses (file offsets) and bytes are printed in hexadecimal and that characters are also printed. If no *file* argument is given, the standard input is read.

Options include:

-s *offset* Specify the beginning offset in the file where printing is to begin. If no *file* argument is given or if a seek fails because the input is a pipe, *offset* bytes are read from the input and discarded. Otherwise, a seek error will terminate processing of the current file.

The *offset* may be given in decimal, hexadecimal (preceded by **0x**), or octal (preceded by a **0**). It is optionally followed by one of the following multipliers: **w**, **l**, **b**, or **k**; for words (2 bytes), long words (4 bytes), blocks (512 bytes), or K bytes (1024 bytes). Note that this is the one case where **b** does *not* stand for bytes. Since specifying a hexadecimal offset in blocks would result in an ambiguous trailing **b**, any offset and multiplier may be separated by an asterisk (*).

-n *count* Specify the number of bytes to process. The *count* is in the same format as *offset* above.

FORMAT FLAGS

Format flags may specify addresses, characters, bytes, words (2 bytes), or longs (4 bytes) to be printed in hexadecimal, decimal, or octal. Two special formats may also be indicated: text or ASCII. Format and base specifiers may be freely combined and repeated as desired to specify different bases (hexadecimal, decimal, or octal) for different output formats (addresses, characters, etc.). All format flags appearing in a single argument are applied as appropriate to all other flags in that argument.

acbwlA

Output format specifiers for addresses, characters, bytes, words, longs, and

ASCII, respectively. Only one base specifier will be used for addresses; the address will appear on the first line of output that begins each new offset in the input.

The character format prints printable characters unchanged, special C escapes as defined in the language, and remaining values in the specified base.

The ASCII format prints all printable characters unchanged, and all others as a period (.). This format appears to the right of the first of other specified output formats. A base specifier has no meaning with the ASCII format. If no other output format (other than addresses) is given, **bx** is assumed. If no base specifier is given, *all* of **xdo** are used.

xdo Output base specifiers for hexadecimal, decimal, and octal. If no format specifier is given, *all* of **acbwl** are used.

t Print a text file, each line preceded by the address in the file. Normally, lines should be terminated by a **\n** character, but long lines will be broken up. Control characters in the range 0x00 to 0x1f are printed as '^@' to '^_'. Bytes with the high bit set are preceded by a tilde (~) and printed as if the high bit were not set. The special characters (^, ~, \) are preceded by a backslash (\) to escape their special meaning. As special cases, two values are represented numerically as '\177' and '\377'. This flag will override all output format specifiers except addresses.

NAME

help - System V Help Facility

SYNOPSIS

help

[help] starter

[help] usage [-d] [-e] [-o] [command_name]

[help] locate [keyword1 [keyword2] ...]

[help] glossary [term]

help arg ...

DESCRIPTION

The System V Help Facility provides on-line assistance for System V users, whether they desire general information or specific assistance for use of the Source Code Control System (SCCS) commands.

Without arguments, *help* prints a menu of available on-line assistance commands with a short description of their functions. The commands and their descriptions are:

COMMAND	DESCRIPTION
starter	information about the System V for the beginning user
locate	locate System V commands using function-related keywords
usage	System V command usage information
glossary	definitions of System V technical terms

The user may choose one of the above commands by entering its corresponding letter (given in the menu), or may exit to the shell by typing **q** (for "quit").

With arguments, *help* directly invokes the named on-line assistance command, bypassing the initial *help* menu. The commands *starter*, *locate*, *usage*, and *glossary*, optionally preceded by the word *help*, may also be specified at shell level. When executing *glossary* from shell level some of the symbols listed in the glossary must be escaped (preceded by one or more backslashes, \ ) to be understood by the Help Facility. For a list of symbols and how many backslashes to use for each, refer to the *glossary*(1) manual page.

From any screen in the Help Facility, a user may execute a command via the shell [*sh*(1)] by typing a ! and the command to be executed. The screen will be

From any screen in the Help Facility, a user may execute a command via the shell [*sh*(1)] by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable **SCROLL** must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your **.profile** file [see *profile*(4)]:

```
export SCROLL ; SCROLL=no
```

If you later decide that scrolling is desired, **SCROLL** must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

The Help Facility can be tailored to a customer's needs by use of the *helpadm*(1M) command.

If the first argument to *help* is different from *starter*, *usage*, *locate*, or *glossary*, *help* assumes information is being requested about the SCCS facility. The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type1 Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (that is, **ge3** for message 3 from the *get* command).

type2 Does not contain numerics (as a command, such as *get*).

type3 Is all numeric (for example, 212).

SEE ALSO

glossary(1), locate(1), sh(1), starter(1), usage(1)
helpadm(1M) in the Administrator's Reference Manual

admin(1), cdc(1), comb(1), delta(1), get(1), prs(1), rmdel(1), sact(1), sccsdiff(1), unget(1), val(1), vc(1), what(1), profile(4), sccsfile(4), term(5) in the Programmer's Reference Manual

WARNINGS

If the shell variable **TERM** [see *sh*(1)] is not set in the user's **.profile** file, then **TERM** will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

[This page left blank.]

NAME

hp - handle special functions of Hewlett-Packard terminals

SYNOPSIS

hp [-e] [-m]

DESCRIPTION

The *hp* command supports special functions of the Hewlett-Packard 2640 series of terminals, with the primary purpose of producing accurate representations of most *nroff* output. A typical usage is in conjunction with DOCUMENTER'S WORKBENCH Software:

nroff -h files ... | hp

Regardless of the hardware options on your terminal, *hp* tries to do sensible things with underlining and reverse line-feeds. If the terminal has the display enhancements feature, subscripts and superscripts can be indicated in distinct ways. If it has the mathematical-symbol feature, Greek and other special characters can be displayed.

The flags are as follows:

-e It is assumed that your terminal has the display enhancements feature, and so maximal use is made of the added display modes. Overstruck characters are presented in the Underlined mode. Superscripts are shown in Half-bright mode, and subscripts in Half-bright, Underlined mode. If this flag is omitted, *hp* assumes that your terminal lacks the display enhancements feature. In this case, all overstruck characters, subscripts, and superscripts are displayed in Inverse Video mode, i.e., dark-on-light, rather than the usual light-on-dark.

-m Requests minimization of output by removal of newlines. Any contiguous sequence of 3 or more newlines is converted into a sequence of only 2 newlines; i.e., any number of successive blank lines produces only a single blank output line. This allows you to retain more actual text on the screen.

With regard to Greek and other special characters, *hp* provides the same set as does *300*(1), except that *not* is approximated by a right arrow, and only the top half of the integral sign is shown.

SEE ALSO

300(1), greek(1)

DIAGNOSTICS

line too long if the representation of a line exceeds 1,024 characters.

The exit codes are **0** for normal termination, **2** for all errors.

BUGS

An overstriking sequence is defined as a printing character followed by a backspace followed by another printing character. In such sequences, if either printing character is an underscore, the other printing character is shown underlined or in Inverse Video; otherwise, only the first printing character is shown (again, underlined or in Inverse Video). Nothing special is done if a backspace is adjacent to an ASCII control character. Sequences of control characters (e.g., reverse line-feeds, backspaces) can make text disappear; in particular, tables generated by *tbl*(1) that contain vertical lines are often missing the lines of text that contain the foot of a vertical line, unless the input to *hp* is piped through *col*(1).

Although some terminals do provide numerical superscript characters, no attempt is made to display them.

NAME

hpio - Hewlett-Packard 2645A terminal tape file archiver

SYNOPSIS

hpio -o [rc] file ...

hpio -i [rta] [-n count **]**

DESCRIPTION

The *hpio* archiver is designed to take advantage of the tape drives on Hewlett-Packard 2645A terminals. Up to 255 System V files can be archived onto a tape cartridge for off-line storage or for transfer to another UNIX system. The actual number of files depends on the sizes of the files. One file of about 115,000 bytes will almost fill a tape cartridge. Almost 300 1-byte files will fit on a tape, but the terminal will not be able to retrieve files after the first 255. This manual page is not intended to be a guide for using tapes on Hewlett-Packard 2645A terminals, but tries to give enough information to be able to create and read tape archives and to position a tape for access to a desired file in an archive.

The *hpio* **-o** command (copy out) copies the specified *files*, together with pathname and status information to a tape drive on your terminal (which is assumed to be positioned at the beginning of a tape or immediately after a tape mark). The left tape drive is used by default. Each *file* is written to a separate tape file and terminated with a tape mark. When *hpio* finishes, the tape is positioned following the last tape mark written.

The *hpio* **-i** command **-i** (copy in) extracts files from a tape drive (which is assumed to be positioned at the beginning of a file that was previously written by an *hpio* **-o**). The default action extracts the next file from the left tape drive.

The *hpio* archiver always leaves the tape positioned after the last file read from or written to the tape. Tapes should always be rewound before the terminal is turned off. To rewind a tape depress the green function button, then function key 5, and then select the appropriate tape drive by depressing either function key 5 for the left tape drive or function key 6 for the right. If several files have been archived onto a tape, the tape may be positioned at the beginning of a specific file by depressing the green function button, then function key 8, followed by typing the desired file number (1-255) with no RETURN, and finally function key 5 for the left tape or function key 6 for the right. The desired file number may also be specified by a signed number relative to the current file number.

The meanings of the available options are:

r Use the right tape drive.

c Include a checksum at the end of each *file*. The checksum is always checked by *hpio* **-i** for each file written with this option by *hpio* **-o**.

n *count* The number of input files to be extracted is set to *count*. If this option is not given, *count* defaults to 1. An arbitrarily large *count* may be specified to extract all files from the tape. The *hpio* archiver will stop at the end of data mark on the tape.

t Print a table of contents only. No files are created. Printed information gives the file size in bytes, the filename, the file access modes, and whether or not a checksum is included for the file.

a Ask before creating a file. The *hpio* **-i** command normally prints the file size and name, creates and reads in the file, and prints a status message when the file has been read in. If a checksum is included with the file, it reports whether the checksum matched its computed value. With this option, the file size and name are printed followed by a **?**. Any response beginning with **y** or **Y** will cause the file to be copied in as above. Any other response will cause the file to be skipped.

FILES

/dev/tty?? to block messages while accessing a tape

SEE ALSO

cu(1C)

DIAGNOSTICS

BREAK
An interrupt signal terminated processing.

Can't create 'file'.
File system access permissions did not allow *file* to be created.

Can't get tty options on stdout.:
The *hpio* archiver was unable to get the input-output control settings associated with the terminal.

Can't open 'file'.
File could not be accessed to copy it to tape.

End of Tape.
No tape record was available when a read from a tape was requested. An

end of data mark is the usual reason for this, but it may also occur if the wrong tape drive is being accessed and no tape is present.

'file' not a regular file.
File is a directory or other special file. Only regular files will be copied to tape.

Readcnt = rc, termcnt = tc.
The *hpio* archiver expected to read *rc* bytes from the next block on the tape, but the block contained *tc* bytes. This is caused by having the tape improperly positioned or by a tape block being mangled by interference from other terminal I/O.

Skip to next file failed.
An attempt to skip over a tape mark failed.

Tape mark write failed.
An attempt to write a tape mark at the end of a file failed.

Write failed.
A tape write failed. This is most frequently caused by specifying the wrong tape drive, running off the end of the tape, or trying to write on a tape that is write protected.

WARNINGS

Tape I/O operations may copy bad data if any other I/O involving the terminal occurs. Do not attempt any type ahead while *hpio* is running. The *hpio* archiver turns off write permissions for other users while it is running, but processes started asynchronously from your terminal can still interfere. The most common indication of this problem, while a tape is being written, is the appearance of characters on the display screen that should have been copied to tape.

The keyboard, including the terminal BREAK key, is locked during tape write operations; the BREAK key is only functional between writes.

The *hpio* archiver must have complete control of the attributes of the terminal to communicate with the tape drives. Interaction with commands such as *cu*(1C) may interfere and prevent successful operation.

BUGS

Some binary files contain sequences that will confuse the terminal.

An *hpio* **-i** that encounters the end of data mark on the tape (for example, scanning the entire tape with *hpio* **-itn 300**), leaves the tape positioned *after* the end of data mark. If a subsequent *hpio* **-o** is done at this point, the data will not

be retrievable. The tape must be repositioned manually using the terminal FIND FILE -1 operation (depress the green function button, function key 8, and then function key 5 for the left tape or function key 6 for the right tape) before the *hpio* **-o** is started.

If an interrupt is received by *hpio* while a tape is being written, the terminal may be left with the keyboard locked. If this happens, the terminal's RESET TERMINAL key will unlock the keyboard.

NAME

iconv - code set conversion

SYNOPSIS

icon -f fromcode -t tocode [-m mode] [-d database] [file(s)]

DESCRIPTION

The *iconv* command converts the encoding of characters in *file(s)* from one code set to another and writes the results to standard output.

The required arguments **fromcode** and **tocode** identify the input and output code sets, respectively. The optional argument **mode** provides a further distinction between mulitple code set maps for the same **fromcode** and **tocode**. The optional argument **database** specifies a database to be used instead of the default database **/usr/lib/kbd/iconv_data**. If no **file(s)** arguments are specified on the command line, *iconv* reads the standard input.

The *iconv* command uses a database with 4 required fields **fromcode, tocode, table, file** and one optional field **mode**. The order of the database fields is as named previously. The database fields are separated by spaces or tabs, and the database rows are separated by newlines.

The *iconv* command matches the required arguments **fromcode** and **tocode** and the optional argument **mode** to the corresponding fields in the database. If a match is found, *iconv* calls *kbdpipe* with the appropriate *table* and *file* fields; i.e., kbdpipe -t *table* -F *file*, where *table* is the mapping between **fromcode** and **tocode** (with optional *mode*), and *file* contains *table*.

The field *mode* does not have to be uniformly included or excluded from the database, i.e. it may be included in some rows and not in others. If the argument **mode** is not included in the *iconv* command line, *iconv* matches the first row found that contains the correct **fromcode** and **tocode** fields, ignoring any *mode* fields.

The naming conventions in the database are left entirely up to the user. However, absolute pathnames are required for *file* fields not located in **/usr/lib/kbd**, as *kbdpipe* assumes that any *file* in the -F *file* argument that does not begin with "/" will be found in **/usr/lib/kbd**.

The codeset conversions supported in the supplied database are given in the following table:

Code Set Conversions Supported			
fromcode	tocode	modes	comment
88591	ascii		
6937	88591	d	Teletext
88591	6937	d	Teletext
646	88591	d b e p	US Ascii
646DE	88591	d	German
646DK	88591	d	Danish
646GB	88591	d	English Ascii
646ES	88591	d	Spanish
646FR	88591	d	French
646IT	88591	d	Italian
646NO	88591	d	Norwegian
646SE	88591	d	Swedish
88591	646	d	7 bit Ascii
88591	646DE	d b e p	German
88591	646DK	d b e p	Danish
88591	646GB	d b e p	English Ascii
88591	646ES	d b e p	Spanish
88591	646FR	d b e p	French
88591	646IT	d b e p	Italian
88591	646NO	d b e p	Norwegian
88591	646SE	d b e p	Swedish
ASCII	88591	d b e	
ASCII	ebcdic	d	
ebcdic	ASCII	d	
ASCII	ibm_ebcdic	d	

The *fromcodes* and *tocodes* 88591, 646, and 6937 correspond to the International Standards ISO 8859-1, ISO 646, and ISO 6937 respectively.

The optional modes, d,b,e,and p have the following meanings:

d default

Any character that cannot be represented is mapped to the ultimate fall back character, which in the tables supplied, is the underscore character (_).

b best fit with no expansion

Characters are, where possible, mapped to the closest approximation of that character but always without expansion, ie., all the character mappings are one-to-one. This will be important, for example, when using curses-based applications where any expansion of a character representation would affect the screen management. [If such code set mappings are performed by the STREAMS-module in the TTY subsystem, then such mappings will be transparent and the application will have no knowledge that these mappings take place.]

e best fit with expansion

Characters of the source code set are, where possible, mapped to the closest approximation of that character in the target code set. Where necessary the character in the source code set is expanded to a sequence of characters in the target code set.

p printer mode - with overstriking

If there is a nondestructive backspace, as exists on many printers, then some characters that are not available can be displayed by overstriking. In this way many accented characters can be displayed.

RETURN VALUES

Returns 0 upon successful completion, 1 otherwise.

EXAMPLES

An example of a database for *iconv* is shown below, with the following fields (field names are not included in the database):

fromcode	tocode	table	filemode
88591	6937	88591.6937.b	pubfilebestfit
88591	6937	pubtable 88591.6937.nt.	normal
646	646DE	togerman /mydir/togerman	

Using the preceding database, the following converts the contents of files **mail1** and **mail2** from code set 88591 to 6937 using *bestfit* mode and stores the results in file **mail.local.**

```
iconv -f 88591 -t 6937 -m bestfit mail1 mail2 > mail.local
```

The following accomplishes the same result as previously shown, as the *bestfit* mode from code set *88591* to *6937* will be the first row containing the correct match.

```
iconv -f 88591 -t 6937 mail1 mail2 > mail.local
```

FILES

/usr/lib/kbd/iconv_data	default database

NOTES

The STREAMS pipe device (**/dev/spx**) and the STREAMS tty subsystem are used by **kbdpipe** and must be installed on the system.

SEE ALSO

kbd(7), kbdcomp(1M), kbdkey(1), kbdmap(1M), kbdpipe(1) in the MNLS 3.2 Product Overview

NAME

ipcrm - remove a message queue, semaphore set, or shared memory id

SYNOPSIS

ipcrm [options]

DESCRIPTION

The *ipcrm* command removes one or more specified messages, semaphore or shared memory identifiers. The identifiers are specified by the following *options*:

-q *msqid* removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.

-m *shmid* removes the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

-s *semid* removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.

-Q *msgkey* removes the message queue identifier, created with key *msgkey*, from the system and destroys the message queue and data structure associated with it.

-M *shmkey* removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

-S *semkey* removes the semaphore identifier, created with key *semkey*, from the system and destroys the set of semaphores and data structure associated with it.

The details of the removes are described in *msgctl*(2), *shmctl*(2), and *semctl*(2). The identifiers and keys may be found by using *ipcs*(1).

SEE ALSO

ipcs(1)
msgctl(2), msgget(2), msgop(2), semctl(2), semget(2), semop(2), shmctl(2), shmget(2), shmop(2) in the Programmer's Reference Manual

[This page left blank.]

NAME

ipcs - report inter-process communication facilities status

SYNOPSIS

ipcs [options]

DESCRIPTION

The *ipcs* command prints certain information about active inter-process communication facilities. Without options, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following options:

-q Print information about active message queues.

-m Print information about active shared memory segments.

-s Print information about active semaphores.

If any of the options **-q**, **-m**, or **-s** are specified, information about only those indicated are printed. If none of these three options are specified, information about all three are printed subject to these options:

-a Use all print *options*. (This is a shorthand notation for **-b**, **-c**, **-o**, **-p**, and **-t**.)

-b Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.) See the following for meaning of columns in a listing.

-c Print creator's login name and group name. See the following text.

-o Print information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments.)

-p Print process number information. (Process ID of last process to send a message and process ID of last process to receive a message on message queues and process ID of creating process and process ID of last process to attach or detach on shared memory segments.) See the following text.

-t Print time information. (Time of the last control operation that changed the access permissions for all facilities. Time of last

msgsnd and last *msgrcv* on message queues, last *shmat* and last *shmdt* on shared memory, last *semop*(2) on semaphores.) See the following text.

-C *corefile*
Use the file *corefile* in place of **/dev/kmem**.

-N *namelist*
The argument is taken as the name of an alternate *namelist* (**/unix** is the default).

-X Print information about XENIX interprocess communication, in addition to the standard interprocess communication status. The XENIX process information describes a second set of semaphores and shared memory. Note that the **-p** option does not print process number information for XENIX shared memory, and the **-t** option does not print time information about XENIX semaphores and shared memory.

The column headings and the meaning of the columns in an *ipcs* listing are given below. The letters in parentheses indicate the options that cause the corresponding heading to appear; **all** means that the heading always appears. Note that these options only determine what information is provided for each facility; they do not determine which facilities are listed.

T (all) Type of facility:
q message queue
m shared memory segment
s semaphore

ID (all) The identifier for the facility entry.

KEY (all) The key used as an argument to *msgget*, *semget*, or *shmget* to create the facility entry. (Note: The key of a shared memory segment is changed to **IPC_PRIVATE** when the segment has been removed until all processes attached to the segment detach it.)

MODE (all) The facility access modes and flags; the mode consists of 11 characters that are interpreted as follows:

The first two characters are:

R if a process is waiting on a *msgrcv*.

S if a process is waiting on a *msgsnd*.

D if the associated shared memory segment has been removed. It disappears when the last process attached to the segment detaches it.

C if the associated shared memory segment is to be cleared when the first attach is executed.

- if the corresponding special flag is not set.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions. The next set refers to permissions of others in the user-group of the facility entry. The last set refers to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is currently unused. Permissions are indicated as follows:

r if read permission is granted
w if write permission is granted
a if alter permission is granted
- if the indicated permission is not granted

OWNER (all)
The login name of the owner of the facility entry.

GROUP(all)
The group name of the group of the owner of the facility entry.

CREATOR (a,c)
The login name of the creator of the facility entry.

CGROUP (a,c)
The group name of the group of the creator of the facility entry.

CBYTES (a,o)
The number of bytes in messages currently outstanding on the associated message queue.

QNUM (a,o)
The number of messages currently outstanding on the associated message queue.

QBYTES (a,b)
The maximum number of bytes allowed in messages outstanding on the associated message queue.

LSPID (a,p)
The process ID of the last process to send a message to the associated queue.

LRPID (a,p)
The process ID of the last process to receive a message from the

associated queue.

STIME (a,t)
The time the last message was sent to the associated queue.

RTIME (a,t)
The time the last message was received from the associated queue.

CTIME (a,t)
The time when the associated entry was created or changed.

NATTCH (a,o)
The number of processes attached to the associated shared memory segment.

SEGSZ (a,b)
The size of the associated shared memory segment.

CPID (a,p)
The process ID of the creator of the shared memory entry.

LPID (a,p)
The process ID of the last process to attach or detach the shared memory segment.

ATIME (a,t)
The time the last attach was completed to the associated shared memory segment.

DTIME (a,t)
The time the last detach was completed on the associated shared memory segment.

NSEMS (a,b)
The number of semaphores in the set associated with the semaphore entry.

OTIME (a,t)
The time the last semaphore operation was completed on the set associated with the semaphore entry.

FILES

/unix	system namelist
/dev/kmem	memory
/etc/passwd	user names
/etc/group	group names

SEE ALSO

msgop(2), semop(2), shmop(2) in the Programmer's Reference Manual

WARNINGS

If the user specifies either the -C or -N flag, the real and effective UID/GID is

set to the real UID/GID of the user invoking *ipcs*.

BUGS

Things can change while *ipcs* is running; the picture it gives is only a close approximation to reality.

[This page left blank.]

NAME

isastream - test for a STREAMS device special file

SYNOPSIS

isastream [device | -]

DESCRIPTION

The *isastream* utility is used to test for a STREAMS device special file. A pathname is given as an argument. Standard input (usually the login terminal) can be specified by a dash character (-).

The specified pathname is opened using the flag O_NDELAY. This means that it will always return control to the calling utility, even if the STREAM is connected to an offline tty device.

The utility uses *isastream*(3C) to do the test and then reports on the result. A usage report is printed in the absence of a pathname.

RETURN VALUES

All output is sent to the standard error file. If the argument is a STREAMS device file then the exit code is 0. In the case of error or a non-STREAMS device file, the exit code is non zero.

EXAMPLES

```
$ isastream /dev/log
/dev/log is a stream
$ isastream -
stdin is a stream
$ isastream /usr/tmp/foo
/usr/tmp/foo is not a stream
```

SEE ALSO

isastream(3C), open(2) in the Programmer's Reference Manual

[This page left blank.]

NAME

join - relational data base operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

The *join* command forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is -, the standard input is used.

File1 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 [see *sort*(1)].

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

The default input field separators are blank, tab, or newline. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the options in the following text use the argument *n*. This argument should be a **1** or a **2** referring to either *file1* or *file2*, respectively. The following options are recognized:

-a*n* 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*.

-j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with 1.

-o *list* Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number. The common field is not printed unless specifically requested.

-t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant. The character *c* is used as the field separator for both input and output.

Internationalization

The *join* command can process characters from supplementary code sets, as well as ASCII characters.

Options:

-e *s* The string *s* to be replaced can contain supplementary characters.

-t *c* The separator *c* can be a character from the supplementary code sets.

EXAMPLE

The following command line will join the password file and the group file, matching on the numeric group ID, and outputting the login name, the group name, and the login directory. It is assumed that the files have been sorted in ASCII collating sequence on the group ID fields.

```
join -j1 4 -j2 3 -o 1.1 2.1 1.6 -t: /etc/passwd /etc/group
```

SEE ALSO

awk(1), comm(1), nawk(1), sort(1), uniq(1)

BUGS

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*, and *awk*(1) are incongruous.

File names that are numeric may cause conflict when the **-o** option is used right before listing filenames.

NAME

kbdcomp - compile kbd tables

SYNOPSIS

kbdcomp [-vrR] [-o *outfile*] [*infile*]

DESCRIPTION

The *kbdcomp* command compiles the tables for use with the *kbd*(7) STREAMS module, a programmable string-translation module. The module has two separate abilities, each of which may be used alone or in combination.

The first ability, "lookup," is that of performing simple substitution of bytes in an input stream. This ability is based on a simple 256-entry lookup table (as there are 256 possible bit combinations for a byte). As input is received, each byte is looked up in the translation table, and the table value for that byte is substituted in place of the original byte. The process is quick, and can be performed on each STREAMS message with no message copying or duplication.

The second ability, "mapping," allows searching for occurrences of specified strings of bytes (or individual bytes) in an input stream, and substituting other strings (or bytes) for them as they are recognized. There are three kinds of mapping that are differentiated by the relationship between the number of bytes in the input and the number of bytes in the output. "One-many" mapping means that for a given byte in the input, many bytes are substituted. "Many-one" includes the other two types as a proper subset, but also includes substitution of many bytes in the input with many bytes of output. The *kbd* compiler can perform all three types of mapping. The lookup ability described in the previous paragraph (i.e. what amounts to "one-one" mapping) is a common special case useful enough to be included separately. By using combinations of both lookup and mapping, a larger class of input translation and conversion problems can be solved than can be solved by the use of either alone.

During operation, processing occurs in two major passes: the lookup table pass always precedes string mapping. The string mapping procedure is nonrecursive for a given table and there is no feedback mechanism (that is, input is scanned in the order received and output is not rescanned for occurrences of recognizable input strings). As an example of mapping, suppose one wishes to translate all occurrences of the string **this** in an input stream into the string **there**. The module recognizes and buffers occurrences of the string **th** (as each byte is received); if the following character is **i**, it will also be buffered, but if **x** is then received, a mismatch is recognized and no translation occurs. Assuming **thi** has been buffered, if the next character seen is **s**, a match is recognized, the buffer containing **this** is discarded, and the string **there** replaces it.

It should be obvious that both input and output strings can be of any non-zero length (see, however, the section on "RESTRICTIONS" in this manpage). Each string to be recognized and translated must be unique, and no complete input string may constitute the leading substring of any other (e.g., you may not define **abc** and **ab** simultaneously, but you may so define **abc**, **agd**, and **abxy**).

Given a filename (or standard input if no name is supplied), *kbdcomp* will compile tables into the output file specified by the **-o** option. If the **-o** option is not supplied, output is to the file **kbd.out**.

The **-v** option causes parsing and verification -- no output file is produced; if no error messages are printed, then the input file is syntactically correct. The **-r** option causes the compiler to check for and report on byte values that cannot be generated in a table (see the description below). The option **-R** is equivalent to **-r**, but it tries to print printable characters as themselves rather than in octal format.

Input Language

Source files for *kbdcomp* are a series of table declarations. Within each table declaration are a number of definitions and functions. A table declaration is one of the forms **map** or **link**:

map *type* (name) {expressions}
link *(string)*

The **link** form will be described in text which follows. The *name* of a map must be a simple token not containing any colons, commas, quotes, or spaces. (For our purposes, a "simple token" is a sequence of alphabetic and/or numeric characters with no embedded punctuation, whitespace, or special symbols.) The *type* field is an optional field that may be either of the keywords, **full** or **sparse**. If omitted, the *type* defaults to sparse. The effect of this field is described in more detail below. The expressions contained in the **map** declaration are one of the following forms. Reserved keywords are printed in constant-width font, variables in italics:

keylist (*string string*)
define (*word value*)
word (*extension result*)
string (*word word*)
strlist (*string string*)
error (*string*)
timed

The **keylist** form is for defining lookup table entries while the remaining forms are the separate string functions.

The definition form (**define**) allows a mnemonic *word* (the first argument) to be associated with a string (the second argument). It is useful for replacing complicated sequences (e.g., those containing special symbols or control characters) with mnemonic words to facilitate the design and readability of tables.

Using the *word* form (where *word* must be a previously defined sequence) in a manner similar to a C function call results in the *value* of *word* being concatenated with *extension*; when the combination is recognized, it is mapped to *result*. The *value* may be a string of characters or a single byte. The following is an illustration (not intended to be complete):

```
map (some_accents) {
        define(acute '\047')
        define(grave '`')
        acute(a '\341')       # same as string("\047a" "\341")
        grave(a '\340')
        # ...et cetera...
        keylist("zyZY" "yzYZ")
}
```

This map defines the single quote and reverse quote keys as "dead-keys" which, when followed by **a**, produce a character from the **ISO 8859-1** code set. It is not necessary for the definition, extension, or result to be a single byte; they may be arbitrary strings.

Strings in definitions and arguments may generally be entered either without quotation or between double quotes. Byte constants may likewise be entered unquoted or between single quotes. The only time quotation is strictly required is when the string contains parentheses, spaces, tab characters, or other special symbols. The language makes no real distinction between byte constants and string constants: both are treated as null-terminated strings; the choice of whether to use a one-character string or a byte constant is thus a matter of taste. Most quoting conventions of C are recognized, except that octal constants must be exactly three digits long. Octal constants may be used in strings as well. In the example above, the arguments to **keylist** need not be quoted, as they contain no special symbols. The following example illustrates some situations where strings must be quoted:

```
string(abc "two words")      # literal space
keylist("[{}]""(())")        # brackets/parentheses
```

```
define(esc_seq"\033\t(")     # tab and parentheses
define(space ' ')            # literal space
string(abc "keylist")        # keyword used as argument
```

Comments in files (inside or outside of map declarations) may be entered in the same manner as for *sh*(1); that is, after a # at the end of a line, or on a line beginning with a #, as shown in the above example.

The keylist form allows single bytes to be mapped to other single bytes; it defines actions that are treated in the lookup table (i.e., are performed before mapping). Any byte value that is not explicitly changed by being included in a keylist form will, of course, be left unchanged; if no keylist forms appear in a map definition, then *kbdcomp* does not generate a lookup table for the map, and the lookup phase is skipped during module operation. Each byte in the first string argument to keylist is mapped to the byte at the same position in the second string argument. That is, given two strings **X** and **Y** as arguments: **Xi** maps to **Yi**, **Xj** maps to **Yj** and so forth. The two arguments must evaluate to strings containing the same number of bytes.

The **string** form has a function similar to mnemonic forms defined with **define** and may be used for any type of many-many mapping. The first argument to **string** is mapped to the second argument (see the comment in the sample map above).

Mappings using both **keylist** and **string** or any **define** forms may be combined. If **i** is mapped to **a** with a **keylist** form, and **a** is used in the sequence **`a**, then when the user types **`i**, the sequence **`a** is seen by the string mapping process (because lookup is done first) and translated accordingly.

The **keylist** form is intended mainly for use in simple keyboard rearrangement and case-conversion applications; **string** is for one-many mapping or for isolated instances of many-many mapping; the **define** form and *word*s defined with it are intended for more general use in groups of related sequences. In some situations while a one-one mapping with keylist may be an obvious choice, the same effect may be achieved with string forms to avoid having a contradictory mapping. For example, suppose one desires, simultaneously, to translate **x** into **y** and **y** into **abc**. If **x** is mapped to **y** by a keylist form and **y** is mapped to **abc** by a string form, then it may be impossible to obtain **y** itself (unless defined in another sequence), even though that was not the intention -- the intention was to obtain **y** whenever the user enters **x**. Following is an example of a contradictory mapping:

```
keylist (x y)
string(y abc) #"y" itself cannot be generated
```

There are cases where the intention is that **y** not be generated, but most often the intention is to generate it. This problem (a relatively common one in code set mapping) can be solved by using a string form to map **x** to **y** initially rather than using a keylist form. This allows both **y** and **abc** to be generated:

```
string(x y)
string(y abc)
```

Entering a large number of one-one mappings with **string** can be somewhat tedious. To make things easier, the **strlist** form is provided. The two string arguments to **strlist** are interpreted in the same manner as arguments to keylist (i.e., they are one-one mappings); however, they are not done by the lookup table, but are processed as string mappings. In the following example, the first three **string** definitions can be reduced to the **strlist** form which follows:

```
string(a b)
string(c d)
string(e f)
strlist(ace bdf)
```

It is important to recognize the difference between **string** and **strlist**: with **string**, the two arguments are a single mapping definition (which may be of any type) whereas with **strlist**, one or more "one-one" string mappings are defined simultaneously. A set of mappings defined with a combination of **string** and **strlist** do not exhibit the same type of incompatibility described previously between keylist and string.

Some further aspects of module processing can now be presented. When a partial match in an input sequence is detected during string processing, it is buffered; if at some point the match no longer succeeds, the first byte of the matched buffer is normally sent to the neighboring module. The rest of the input is left in the buffer and scanned again to see if it matches the beginning of another sequence. The **error** entry allows one to send a string (or byte) constant (called a "fallback" character) instead of the byte that began the previous sequence; this is particularly useful in code set mapping and conversion applications where the character which failed to be translated might be one which does not occur or has some other meaning in the target code set. The

following example illustrates use of the **error** format:

```
#turn arrow keys into vi commands
map(vi_map) {
        string("\033[A" k)    # up
        string("\033[B" j)    # down
        error("!")
}
```

Given input of the *escape* character followed by [A or [B, a single character (j or k) is generated. If presented with the sequence *escape*-[Q, the module will produce the sequence ![Q. The error string ! replaces *escape* because the sequence failed to match when Q was received. The remaining characters are rescanned, and neither [nor Q is found to begin a recognized sequence.

One-one mapping with strings or other defined forms (rather than by a keylist lookup table) is generally performed with a linear search operation when looking for bytes which begin sequences. However, if the table is specified as a full table, it is initially indexed rather than searched linearly and, thus, processed much more quickly when there are a large number of entries. This should be kept in mind in code set mapping applications where nearly all characters are mapped, and many (or most) are one-one mappings. If only a very few characters are mapped with string functions, one must decide whether to trade a small gain in processing speed for the space needed to store the index if a table is made full.

The **link** form is used to produce a **composite** table. A composite table is really a form of linkage that allows several tables to be used together in sequence as if the sequence were a single table. The string argument to **link** is of the following form:

```
composite: component1, component2, componentn
```

The target *composite* name is followed by a colon, and the ordered *component* list is comma-separated. If the string argument contains spaces or special characters, it must be quoted. (This string is not interpreted by *kbdcomp*, but is left intact in the output field; it is interpreted by the module at runtime.) When a **composite** table is used, the effect is similar to pushing more than one instance of the *kbd*(7) module in the sense that the component tables function sequentially, but it is accomplished within a single instance of the module. As output is produced by processing with one table in the composite, the data is subsequently processed by the next component and so forth until the final result emerges at the end of the sequence. (There is no restriction on the use of any combination of full and sparse tables in *composite*.)

Composite tables are useful for simplifying complex mapping situations by modularizing the processing and for increasing the reusability of tables for different mapping applications. Tables primarily implementing code set mappings may be linked to other tables primarily implementing compose- or dead-key sequences. With a single table implementing a common code set mapping, several different tables implementing combinations of code set mapping and compose-key layouts may be built. A typical configuration might use one table for mapping from an external to internal code set, then use one or more separate tables working in the internal code set to provide compose- or dead-key functionality, as in the following example. One table, **646Sp-8859**, maps from an ISO 646 variant (Spanish) external code set to **ISO 8859 1**; this is combined with two other tables respectively implementing **8859-1** by compose-sequences, and by dead-key sequences:

```
link("composed:646Sp-8859,8859-1-cmp")
link("deadkey:646Sp-8859,8859-1-dk")
```

Composite tables can also be built while the module is running from the *kbdload*(1M) command line; details are in the *kbdload*(1M) manual page. The component tables are linked and processed in the given order (left-to-right). Because the link argument is actually parsed at runtime [by *kbd*(7)], it is not an error to refer to tables that are not contained in the file currently being compiled. An error will be generated when the file is loaded if any component of a link is not present in memory at that time.

The directive **timed** may appear any place within a **map** definition. If used, it causes the table in which it is defined to be interpreted in timeout mode. In this mode, string mappings are considered to not match if more than a certain amount of time elapses after receipt of the first byte of a sequence without its being fully received and mapped. Given a timed map in which **abc** is to be mapped to **xyz** and the timeout value is **30**, if the user types **ab**, then waits for longer than 30 time units before typing **c**, the entire sequence will not be translated. In this case, the sequence is treated as any other mismatch would be: **a** is passed to the neighboring module, and **b** is checked to see if it begins a sequence. The timer is reset when a mismatch occurs, so that if **bc** is defined in this situation and **c** has just been received, it will be mapped as expected. The default timeout is typically 1/5 to 1/3 of a second (see *kbd*(7) for details).

Timeout mode is generally useful in situations where terminal function keys are being interpreted, to distinguish between a string typed by the user and a function key string sent by the terminal; it is not intended for use with "batch" applications such as *kbdpipe*(1M). In a **composite** table, some components may be timed and some not, making the mode useful for combinations of code set

mapping and function key mapping.

Timing depends on several factors, including terminal baud-rate, system load, and the user's typing speed. If the timeout value is too long, then typed sequences that happen to be the same as function keys will be erroneously mapped; if the value is too short, then function keys may be missed under a heavy system load or with low speed devices. See *kbdset(1)* for information on how to change the timeout value, and *kbd*(7) for information on how an administrator may change the default timeout value. This directive should never be used in tables that implement code set mapping, as it makes the results quite unpredictable. Long timeouts, on the order of seconds, may be useful in some contexts.

Building & Debugging

Users who intend to build their own tables may study the source tables supplied with the distribution in the directory **/usr/lib/kbd/src**.

If characters other than alphanumerics are to be used, quoted strings are preferred to unquoted strings; quotation is required for some characters, as mentioned previously. Map names and the first arguments of **define** should be alphanumeric tokens.

The report generated by the **-r** option may be useful for debugging complex tables. The report (produced on standard error) consists of two octal lists. One list contains byte values that cannot be generated from the lookup table (if keylist forms are used). The other list contains byte values that cannot be generated in any way; in other words, values that are neither parts of "result text" (i.e.., products of string mappings) nor generated by the lookup table (if there is one), but that are used in other sequences. The report does not exhaustively list unreachable paths, but may indicate whether they exist and help pinpoint them.

Output Files

The files produced by *kbdcomp* begin with a header. The magic string is **kbd!map**, with a version number. This header is immediately followed by the tables themselves. (A file can contain more than one table.) The lines below can be added to the **/etc/magic** file for the *file*(1) command to recognize **kbd** files.

```
0       string      kbd!map         kbd map file
>8      byte        >0              Ver %d:
>10     short       >0              with %d table(s)
```

RESTRICTIONS

A maximum length of 128 bytes for input strings and 256 bytes for output strings is imposed. The total amount of space consumed by a single table is limited to

around 65,000 bytes. Versions are strictly incompatible: "object" tables are machine-dependent in their byte order and structure size. Thus, while source files are portable, the output of *kbdcomp* is not. This implies that when using remote devices across a network between heterogeneous machines, tables must be loaded on the machine where the module is actually pushed (i.e., the remote side).

FILES

/usr/lib/kbd	directory containing system standard map files
/usr/lib/kbd/src	source for some system map files

SEE ALSO

kbdset(1)
kbdload(1M), kbd(7) in the Administrator's Reference Manual

[This page left blank.]

NAME

kbdpipe - use the kbd module in a pipeline

SYNOPSIS

kbdpipe **-t** table [**-f** tablefile] [**-F**] [**-o** outfile] [infile(s)]

DESCRIPTION

The program allows the use of *kbd*(7) tables as pipeline elements between user programs. [General descriptions of the module and its capabilities appear in *kbdcomp*(1M) and *kbd(7)*.] The *kbdpipe* command is mostly useful in code set conversion applications. If an *output* file is given, then all *infiles* are piped to the given *output* file. With no arguments other than **-t**, standard input is converted and sent to standard output.

The required **-t** option argument identifies the *table* to be used for conversion. If the *table* has already been loaded as a shared table [see *kbdload*(1M)], it is attached. If, however, the table has not been loaded, an attempt is made to load it. If the given table name is not an absolute pathname, then the name of the system mapping library (**/usr/lib/kbd**) is prepended to the argument, and an attempt is made to load the table from the resulting pathname [i.e., it becomes an argument to the loader, *kbdload*(1M)]. Assuming the table can be loaded, it is attached.

The argument to **-f** defines the filename from which the table will be loaded, overriding the default action described above. The file is loaded (in its entirety), and the named table attached. This option should be used if the default action would fail.

The output file specified by **-o** must not already exist (a safety feature.) The option **-F** may be used to override the check for existence of the output file; in this case, any existing *outfile* will be truncated before being written.

EXAMPLES

The following example converts two input files into relative nonsense by mapping ASCII into Dvorak keyboard equivalents using the Dvorak table. The table is assumed to reside in the file **/usr/lib/kbd/Dvorak**. The existing output file is forcefully overwritten:

```
kbdpipe -F -t Dvorak -o iapxai.vj file1 file2
```

The following example loads the Dvorak table from a different file, then converts standard input to standard output. The *Dvorak* table (assumed to be nonresident) is explicitly loaded from an absolute path beginning at the user's home directory:

```
kbdpipe -t Dvorak -f $HOME/tables/Dvorak.tab
```

FILES

/usr/lib/kbd - directory containing system standard table files

SEE ALSO

kbdset(1)
kbd(7), kbdload(1M) in the Administrator's Reference Manual

WARNINGS

Because *kbdpipe* uses *kbdload*(1) to load tables, it cannot resolve link references. Therefore, if a composite table is to be used, then the relevant portions must either be already loaded and public, or be contained in the file indicated (using the **-f** option) on the command line. In this case, the composite elements must be loaded earlier than the link entry.

NAME

kbdset - attach to kbd mapping tables, set modes

SYNOPSIS

kbdset [-oq] [-{a|d} table] [-v string] [-k hotkey] [-m x] [-t ticks]

DESCRIPTION

The normal user interface to the *kbd*(7) module is *kbdset*. (See *kbdcomp*(1M) and *kbd*(7) for a general description of the module's capabilities.) The *kbdset* interface allows users to attach to preloaded tables, detach from tables, and to set options. Options are provided for setting "hot keys" to toggle tables and for controlling modes of the module.

Arguments and options are scanned and acted upon in command line order. If the **-o** option is given, subsequent options affect the output side of the Stream; otherwise the input side is assumed.

Presence of the **-q** option causes *kbdset* to list modules which can be accessed by the invoking user. In this case all subsequent options are ignored. The output from the **-q** option lists the user's current hot key settings; current timer value; and, for each available table, an identifier, the name, size, attachments (input and/or output sides), reference count, number of components, and type (private or public). In the following example, there is one composite table, two tables are attached on the input side, and one on the output side:

```
In Hot Key = ^_
Timers: In = 20; Out = 20
```

ID Name	Size	I/O	Ref	Cmp	Type
40319800 Deutsche	324	i -	4	-	pub
4033de80 Case/Dvorak [40316800] [40316a00]	68	- -	0	2	pri
40316800 Case	312	- o	1	-	pri
40316a00 Dvorak	312	i -	1	-	pri

The ID field is an identifier unique to a given table (actually its address in memory). Currently attached tables are marked **i** or **o**, otherwise the I/O fields are marked with a dash. *Ref* is a reference count of attached users (including composites that refer to simple tables) and, if non-zero, indicates that the table is "in use". *Size* is the total size in bytes of the table and associated overhead in memory. If the table is a composite table, the *Cmp* field contains a number instead of a dash, and the following line lists an *identifier* for each component, in order of processing (allowing identification of the components in a composite table). Publicly available tables are marked with the type **pub** and private tables

with **pri**. Private tables are available only to the invoking user and within the current Stream. Tables that are interpreted in timeout mode [see *kbdcomp*(1M)] have an asterisk (*) preceding the *type* field; members of composite tables that are interpreted in timeout mode have an asterisk after their bracketed identifier (on the second output line).

The option **-a** accompanied by an argument attaches to the named table. A table may not be multiply attached by a single user. When a table is attached and no other table is already attached, then the table is automatically made current. The option **-d** detaches from the named table. (See *kbdload*(1M) for a description of how tables are loaded.)

The **-k** option sets the user's hot key. Setting a hot key with only a single active table allows mapping to be toggled on and off, depending on the hot key mode. A hot key is a single byte, typically set to a relatively unused control character, that is caught by the *kbd* module and used for module control rather than being translated in any way. The key used as a hot key becomes unavailable for other uses (unless it is generated by mapping!). The hot key may be reset at any time, independently from other options.

The **-m** option with an integer argument controls the hot key mode. Legal modes are **0, 1** (the default), and **2**. Mode **0** allows one to toggle through the list of attached tables. Upon reaching the end of the list, the cycle returns to the beginning of the list. Use of Mode **0** with only one table loaded does not allow mapping to be turned off. Mode **1** toggles to the unmapped state upon reaching the end of the list (e.g., given two tables, the sequence is *table1, table2, off, table1*, etc.). Mode **2** toggles to the unmapped (or off) state between every table in the list of attached tables (e.g., given two tables, the sequence is *table1, off, table2, off, table1*, etc.).

The **-v** option turns on "verbose" mode, which can be useful when multiple tables are used in interactive sessions. In verbose mode, the name of the table can be output to the terminal whenever the user changes to a new table with the hot key. The string associated with the option can be any short string. If the character sequence *%n* appears in the string, the name of the current table (or a null string) will be substituted for the *%n*. (A null argument to **-v** is equivalent to "terse" mode.) One useful sequence for this mode is *save-cursor, goto-status-line, clear-to-end-of-line, "%n", restore-cursor*. This causes output of the current table name on the terminal's status line. (See the *terminfo*(4) description for the appropriate escape sequences.) Verbose mode is only available to show input table status to the output side of the Stream. The output string for verbose mode is not passed through the mapping process, but is transmitted directly downstream with no other interpretation (it should thus be a string of ASCII

characters or in some other externally available code set).

The **-t** option with an argument is used to change the timer for tables in the Stream that are interpreted in timeout mode. Values (in "clock ticks") between 5 and 400 are acceptable. (Depending on the hardware, the clock is usually either 60Hz or 100Hz; thus, one tick is either 1/60 or 1/100 of a second. With some experimentation, a suitable value for one's own system and typing speed can be found.) When a table that uses timeout mode is attached, it is assigned the current timer value. All tables that are attached after setting the timer value will take on the new value, but tables currently attached are unaffected (this allows one to set different values for different tables). The option does not affect other users' values. The timer value may be set independently for input and output sides by using **-t** in conjunction with f3-o. The value for a currently attached table may be reset by detaching the table, setting the value, then reattaching the table.

In the *query* output, the line beginning with **Timers:** shows the timer values for input and output sides of the module.

FILES

/usr/lib/kbd - directory containing system standard map files

SEE ALSO

kbdcomp(1M), kbdload(1M), kbd(7) in the Administrator's Reference Manual

BUGS

It is not possible with the **-q** option to see the timer values assigned to currently attached tables, nor to reset the value for a table that is currently attached.

Better control of timeout mode and values should be provided in the future.

RESTRICTIONS

A table may be detached while it is current; however, in this case, it is first made non-current; this allows error recovery under adverse circumstances.
Detachment of a current table is not affected by the current hot key mode, but always toggles to a state where no table is current.

[This page left blank.]

NAME

keepopen - open a file and keep it open

SYNOPSIS

keepopen [file | -]

DESCRIPTION

The *keepopen* utility is used to hold a file open. This is particularly useful for holding open STREAMS device special files. A STREAM is normally dismantled on the last close, so this utility ensures that the STREAM is opened and held intact.

The STREAM to be kept open is specified by *file*, or dash character (-) for standard input. The utility returns immediately leaving a child process sleeping with *file* open. This process can be killed by any signal except SIGINT and SIGQUIT, in which case the STREAM is closed.

RETURN VALUES

If the *file* does not exist or has incorrect access permission then *keepopen* exits with a return value of 1. On successful completion, the parent process will exit with return value 0. The child process has an indeterminate return value. All output is sent to standard error.

EXAMPLES

If the *nls*(7) module were required for output to a printer, then *keepopen* can be used to open the STREAMS device special file for the printer and then *nls* can be pushed. The configuration is then available for all subsequent accesses to the printer.

```
# keepopen printer
printer is open
# strpush printer nls
pushed module nls on printer
# strlook printer
the top module on printer is nls
```

SEE ALSO

strpush(1), strlook(1), kill(1)
signal(2) in the Programmer's Reference Manual

[This page left blank.]

NAME

kill - terminate a process

SYNOPSIS

kill [-signo] PID ...

DESCRIPTION

The *kill* command sends signal 15 (terminate) to the specified processes. This normally kills processes that do not catch or ignore the signal. The process number of each asynchronous process started with & is reported by the shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using *ps*(1).

The details of the kill are described in *kill*(2). For example, if process number 0 is specified, all processes in the process group are signaled.

The killed process must belong to the current user unless he is the superuser.

If a signal number preceded by - is given as first argument, that signal is sent instead of terminate [see *signal*(2)]. In particular, kill -9 ... is a sure kill.

SEE ALSO

ps(1), sh(1)
kill(2), signal(2) in the Programmer's Reference Manual

[This page left blank.]

NAME

last - indicate last logins of users and teletypes

SYNOPSIS

last [-N] [name ...] [tty ...]

DESCRIPTION

The *last* command prints the sessions of the specified users and teletypes, most recent first, indicating the time the session began, the duration of the session, and the teletype which the session took place on. The *last* command will indicate if the session is still continuing or was cut short by a reboot.

The *last* command prints information from the *wtmp* file which records all logins and logouts for information about a user, a teletype or any group of users and teletypes.

If *last* is interrupted, it indicates how far the search has progressed in *wtmp*.

Control-d (EOF) signal does nothing. Therefore, exit gracefully from *last* with a break or shift/delete signal.

ARGUMENTS

Arguments specify names of users or teletypes of interest.

Names of teletypes may be given fully or abbreviated. For example **last 00** is the same as **last tty00.**

If multiple arguments are given, *last* prints the information which applies to any of the arguments. For example:

```
last root ttyb
```

would list all of the sessions of **root** as well as all sessions on the terminal **ttyb.**

The *last* command, with no arguments, prints a record of all logins and logouts in reverse order. The -N option limits the report to N lines.

EXAMPLE

The pseudo-user **reboot** logs in at reboots of the system, thus

```
last reboot
```

gives an indication of mean time between reboots.

FILES

/etc/wtmp	login data base

SEE ALSO

utmp(4)

NAME

line - read one line

SYNOPSIS

line

DESCRIPTION

The *line* command copies one line (up to a newline) from the standard input and writes it on the standard output. It returns an exit code of 1 on EOF and always prints at least a newline. It is often used within shell files to read from the user's terminal.

SEE ALSO

sh(1)
read(2) in the Programmer's Reference Manual

[This page left blank.]

NAME

locate - identify a System V command using keywords

SYNOPSIS

[help] locate

[help] locate [keyword1 [keyword2] ...]

DESCRIPTION

The *locate* command is part of the System V Help Facility, and provides on-line assistance with identifying System V commands.

Without arguments, the initial *locate* screen is displayed from which the user may enter keywords functionally related to the action of the desired System V commands they want identified. A user may enter keywords and receive a list of System V commands whose functional attributes match those in the keyword list, or may exit to the shell by typing q (for "quit"). For example, if you want to print the contents of a file, enter the keywords "print" and "file". The *locate* command would then print the names of all commands related to these keywords.

Keywords may also be entered directly from the shell, as shown above. In this case, the initial screen is not displayed, and the resulting command list is printed.

More detailed information on a command in the list produced by *locate* can be obtained by accessing the *usage* module of the System V Help Facility. Access is made by entering the appropriate menu choice after the command list is displayed.

From any screen in the Help Facility, a user may execute a command via the shell [*sh*(1)] by typing a ! and the command to be executed. The screen will be redrawn if the command that was executed was entered at a first level prompt. If entered at any other prompt level, only the prompt will be redrawn.

By default, the Help Facility scrolls the data that is presented to the user. If you prefer to have the screen clear before printing the data (non-scrolling), the shell variable **SCROLL** must be set to **no** and exported so it will become part of your environment. This is done by adding the following line to your **.profile** file [see *profile*(4)]:

```
export SCROLL ; SCROLL=no
```

If you later decide that scrolling is desired, **SCROLL** must be set to **yes**.

Information on each of the Help Facility commands (*starter*, *locate*, *usage*, *glossary*, and *help*) is located on their respective manual pages.

SEE ALSO

glossary(1), help(1), sh(1), starter(1), usage(1)
term(5) in the Programmer's Reference Manual

WARNINGS

If the shell variable **TERM** [see *sh*(1)] is not set in the user's **.profile** file, then **TERM** will default to the terminal value type 450 (a hard-copy terminal). For a list of valid terminal types, refer to *term*(5).

NAME

login - sign on

SYNOPSIS

login [name [environment-variable ...]]

DESCRIPTION

The *login* command is used at the beginning of each terminal session and allows you to identify yourself to the system. It may be invoked as a command or by the system when a connection is first established. Also, it is invoked by the system when a previous user has terminated the initial shell by typing a <Ctrl> d to indicate an end-of-file.

If *login* is invoked as a command it must replace the initial command interpreter. This is accomplished by typing:

```
exec login
```

from the initial shell.

The *login* command asks for your user name (if not supplied as an argument), and, if appropriate, your password. Echoing is turned off (where possible) during the typing of your password, so it does not appear on the written record of the session.

If you make a mistake in the login procedure you will receive the message

```
Login incorrect
```

and a new login prompt does appear. If you make five incorrect login attempts, all five may be logged in **/usr/adm/loginlog** (if it exists) and the line is dropped.

If you do not complete the login successfully within a certain period of time (e.g., one minute), you are likely to be silently disconnected.

After a successful login, the user ID, the group ID, the working directory, and the command interpreter [usually *sh*(1)] is initialized. If the shell **/bin/sh** is running, accounting files are updated, the procedure **/etc/profile** is performed, the message of the day (if any) is printed, and the file **.profile** in the working directory is executed, if it exists. If the shell **/bin/csh** is running, the **.login** and **.cshrc** files in the working directory are executed, if they exist. These specifications are found in the **/etc/passwd** file entry for the user. The name of the command interpreter is - followed by the last component of the interpreter's pathname (i.e., **-sh**). If this field in the password file is empty, then the default command interpreter, **/bin/sh** is used. If this field is *, then the named directory becomes the root directory, the starting point for path searches for pathnames

beginning with a /. At that point, *login* is executed again at the new level which must have its own root structure, including **/etc/login** and **/etc/passwd.**

The basic environment is initialized to:

HOME=*your-login-directory*
PATH=:/bin:/usr/bin
SHELL=*last-field-of-passwd-entry*
MAIL=/usr/mail/*your-login-name*
TZ=*timezone-specification*

The environment may be expanded or modified by supplying additional arguments to *login*, either at execution time or when *login* requests your login name. The arguments may take either the form *xxx* or *xxx=yyy*. Arguments without an equal sign are placed in the environment as

L*n*=xxx

where *n* is a number starting at 0 and is incremented each time a new variable name is required. Variables containing an = are placed into the environment without modification. If they already appear in the environment, then they replace the older value. There are exceptions. The variables **HOME, PATH, SHELL, MAIL, IFS, TZ, HZ, CDPATH,** and **LOGNAME** cannot be changed. This prevents people, logging into restricted shell environments, from spawning secondary shells which are not restricted. Both *login* and *getty* understand simple single-character quoting conventions. Typing a backslash in front of a character quotes it and allows the inclusion of such things as spaces and tabs.

FILES

/etc/utmp	accounting
/etc/wtmp	accounting
/usr/mail/*your-name*	mailbox for user *your-name*
/usr/adm/loginlog	record of failed login attempts
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/profile	system profile (**/bin/sh** only)
.profile	user's login profile (**/bin/sh** only)

NOTES

The file **/etc/default/login** contains special login information including the flag PASSREQ. When PASSREQ is set to YES (PASSREQ=YES), users will be required to have a password. When a user without a password logs in and PASSREQ=YES, the user will be forced to add a password before the user is allowed access to the system. One exception to this requirement is if password

aging is turned on for the user and the NULL password has not been aged. In this case, the user will be allowed to access the system without a password until the NULL password has been aged, or until the root user forces the password to be aged (*passwd -f* command).

SEE ALSO

mail(1), sh(1)
newgrp(1M), su(1M) in the Administrator's Reference Manual
loginlog(4), passwd(4), profile(4), environ(5) in the Programmer's Reference Manual

DIAGNOSTICS

login incorrect if the user name or the password cannot be matched.

No shell, cannot open password file, or *no directory*: consult a UNIX system programming counselor.

No utmp entry. You must *exec* login from the lowest level shell if you attempted to execute *login* as a command without using the shell's *exec* internal command or from other than the initial shell.

[This page left blank.]

NAME

logname - get login name

SYNOPSIS

logname

DESCRIPTION

The *logname* command returns the contents of the environment variable **$LOGNAME**, which is set when a user logs into the system.

FILES

/etc/profile

SEE ALSO

env(1), login(1)
logname(3X), environ(5) in the Programmer's Reference Manual

[This page left blank.]

NAME

look - find lines in a sorted list

SYNOPSIS

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)

RESTRICTIONS

Some special characters, such as parentheses, ampersands, and carats, are shell sensitive and therefore are not recognized as strings. These characters may be used, however, if preceded by a backslash.

[This page left blank.]

NAME

lp, cancel - send/cancel requests to an LP print service

SYNOPSIS

lp [*printing options*] *files*
lp -i *id printing options*
cancel [*ids*] [*printers*]

DESCRIPTION

The first form of the *lp* shell command arranges for the named files and associated information (collectively called a *request*) to be printed. If no filenames are specified on the shell command line, the standard input is assumed. The standard input may be specified along with named *files* on the shell command line using the filename. The *files* are printed in the order they appear on the shell command line.

The second form of *lp* is used to change the options for a request. The print request identified by the *request-id* is changed according to the printing options specified with this shell command. The printing options available are the same as those with the first form of the *lp* shell command. If *request-id* has finished printing, the change is rejected. If the *request-id* is already printing, it is stopped and restarted from the beginning, unless the **-P** option has been given.

The *lp* command associates a unique *id* with each request and prints it on the standard output. This *id* can be used later to cancel, change, or find the status of the request. (See the section on *cancel* for details about canceling a request, the previous paragraph for an explanation of how to change a request, and *lpstat*(1) for information about checking the status of a print request.)

Sending a Print Request

The first form of the *lp* command is used to send a print request to a particular printer or group of printers.

Options to *lp* must always precede filenames but may be listed in any order. The following options are available for *lp*:

-c Makes copies of the *files* to be printed immediately when *lp* is invoked. Normally, *files* will not be copied. If the **-c** option is not given, then the user should be careful not to remove any of the *files* before the request has been printed in its entirety. It should also be noted that in the absence of the **-c** option, any changes made to the named *files* after the request is made but before it is printed will be reflected in the printed output.

-d *dest* Prints this request using *dest* as the printer or class of printers. Under certain conditions (lack of printer availability, capabilities of printers, and so on), requests for specific destinations may not be accepted [see *accept*(1M) and *lpstat*(1)]. By default, *dest* is taken from the environment variable **LPDEST** (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems [see *lpstat*(1)].

-f *form-name* **[-d any]**
Prints the request on the form *form-name*. The LP print service ensures that the form is mounted on the printer. If *form-name* is requested with a printer destination that cannot support the form, the request is rejected. If *form-name* has not been defined for the system or if the user is not allowed to use the form, the request is rejected [see *lpforms*(1M)]. When the **-d any** option is given, the request is printed on any printer that has the requested form mounted and can handle any other needs of the print request.

-H *special-handling*
Prints the request according to the value of *special-handling*. Acceptable values for *special-handling* are **hold**, **resume**, and **immediate**, as defined in the following text:

hold Will not print the request until notified. If already printing, stops it. Other print requests will go ahead of a held request until it is resumed.

resume Resumes a held request. If it had been printing when held, it will be the next request printed, unless subsequently bumped by an **immediate** request.

immediate
(Available only to LP administrators)
Prints the request next. If more than one request is assigned **immediate**, the requests are printed in the reverse order queued. If a request is currently printing on the desired printer, you have to put it on hold to allow the immediate request to print.

-m Sends mail [see *mail*(1)] after the files have been printed. By default, no mail is sent upon normal completion of the print request.

-n *number*
Prints *number* copies of the output. (Default is 1.)

-o *option*

Specifies printer-dependent or class-dependent *options*. Several such *options* may be collected by specifying the **-o** keyletter more than once. The standard interface recognizes the following options:

nobanner

Does not print a banner page with this request. (The administrator can disallow this option at any time.)

nofilebreak

Does not insert a form feed between the files given if submitting a job to print more than one file.

length=*scaled-decimal-number*

Prints the output of this request with pages *scaled-decimal-number* lines long. A *scaled-decimal-number* is an optionally scaled decimal number that gives a size in lines, columns, inches, or centimeters, as appropriate. The scale is indicated by appending the letter "i" (for inches) or the letter "c" (for centimeters). For length or width settings, an unscaled number indicates lines or columns; for line pitch or character pitch settings, an unscaled number indicates lines per inch or characters per inch (the same as a number scaled with "i"). For example, **length=66** indicates a page length of 66 lines, **length=11i** indicates a page length of 11 inches, and **length=27.94c** indicates a page length of 27.94 centimeters.

This option cannot be used with the **-f** option.

width=*scaled-decimal-number*

Prints the output of this request with page-width set to *scaled-decimal-number* columns wide. (See the explanation above for *scaled-decimal-numbers*.) This option cannot be used with the **-f** option.

lpi=*scaled-decimal-number*

Prints this request for "lines per inch" with the line pitch set to *scaled-decimal-number* lines per inch. This option cannot be used with the -f option.

cpi=*scaled-decimal-number*

Prints this request for "characters per inch" with the character pitch set to *scaled-decimal-number* characters per inch. Character pitch can also be set to **pica** (representing 10 columns

per inch) or **elite** (representing 12 columns per inch), or it can be **compressed**, which is as many columns as a printer can handle. There is no standard number of columns per inch for all printers; see the *terminfo*(4) database for the default character pitch for your printer. The **cpi** option cannot be used in conjunction with the **-f** option.

stty=*stty-option-list*
A list of options valid for the **stty** command. Enclose the list with quotes if it contains blanks.

-P *page-list*
Prints the page specified in *page-list*. This option can be used only if there is a filter available to handle it; otherwise, the print request will be rejected.

The *page-list* may consist of range(s) of numbers, single page numbers, or a combination of both. The pages will be printed in ascending order.

-q *priority-level*
Assigns this request *priority-level* in the printing queue. The values of *priority-level* range from 0, the highest priority, to 39, the lowest priority. If a priority is not specified, the default for the print service is used, as assigned by the system administrator.

-s Suppresses messages from *lp*(1) such as "request id is ...".

-S *character-set* **[-d any]**
-S *print-wheel* **[-d any]**
Prints this request using the specified *character-set* or *print-wheel*. If a form has been specified that requires a *character-set* or *print-wheel* other than the one specified with the **-S** option, the request is rejected.

For printers that take print wheels, if the *print-wheel* specified is not one listed by the administrator as acceptable for the printer involved in this request, the request is rejected unless the print wheel is already mounted on the printer. For printers that use selectable or programmable character sets, if the *character-set* specified is not one defined in the Terminfo database for the printer [see *terminfo*(4)] or is not an alias defined by the administrator, the request is rejected.

When the **-d any** option is used, the request is printed on any printer that has the print wheel mounted or any printer that can select the character set and can handle any other needs of the request.

-t *title* Prints *title* on the banner page of the output. The default is no title.

-T *content-type* [-r]
Prints the request on a printer that can support the specified *content-type*. If no printer accepts this type directly, a filter will be used to convert the content into an acceptable type. If the -r option is specified, a filter will not be used. If -r is specified but no printer accepts the *content-type* directly, the request is rejected. If the *content-type* is not acceptable to any printer, either directly or with a filter, the request is rejected.

-w Writes a message on the user's terminal after the *files* have been printed. If the user is not logged in, then mail will be sent instead.

-y *mode-list*
Prints this request according to the printing modes listed in *mode-list*. The allowed values for *mode-list* are locally defined. This option can be used only if there is a filter available to handle it; if there is no filter, the print request will be rejected.

Canceling a Print Request

The *cancel* command cancels printer requests that were made by the *lp*(1) shell command. The shell command line arguments may be either *request-ids* [as returned by *lp*(1)] or *printer* names [for a complete list, use *lpstat*(1)]. Specifying a *request-id* cancels the associated request even if it is currently printing. Specifying a *printer* cancels the request that is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

Internationalization

The *lp* command sends files or standard input containing characters from supplementary code sets to an LP line printer.

FILES

/usr/spool/lp/*

NOTES

Printers for which requests are not being accepted will not be considered when the destination is any. (Use the *lpstat* -a command to see which printers are accepting requests.) On the other hand, if a request is destined for a class of printers and the class itself is accepting requests, *all* printers in the class will be considered, regardless of their acceptance status, as long as the printer class is accepting requests.

SEE ALSO

enable(1), lpstat(1), mail(1)
accept(1M), lpadmin(1M), lpfilter(1M), lpforms(1M), lpsched(1M), lpusers(1M) in the Administrator's Reference Manual
terminfo(4) in the Programmer's Reference Manual

WARNINGS

For printers that take mountable print wheels or font cartridges, if you do not specify a particular print wheel or font with the **-S** option, whatever printwheel happens to be mounted at the time your request prints will be used. Use the *lpstat* **-p -l** command to see what print wheels are available. For printers that have selectable character sets, you will get the standard set if you don't use the **-S** option.

RESTRICTIONS

Option:

-t *title* Supplementary characters specified in *title* do not print correctly. Refer to *banner*(1).

NAME

lps - set parallel printer characteristics

SYNOPSIS

lps *device name* [*lines columns indent*]

DESCRIPTION

The *lps* command allows the user to change the line, column, and indentation settings of printers connected to a parallel interface. When used without options, *lps* reports the current settings for the specified device.

The three options may not be set separately; all must be specified at once. Also, *lps* must be used after every reboot or system reset if you want to keep the designated changes.

Lines is lines per page; *columns* is columns per line (to the right of the current indent); and *indent* is the number of columns from the left margin to the first column of print.

EXAMPLES

The following line lists the current settings for device Pdoc:

```
lps Pdoc
```

The following line would set printer Psp to 66 lines per page, 132 columns per page, and 0 indent:

```
lps Psp 66 132 0
```

DIAGNOSTICS

Can't open *device name* *lps* could not access the device specified.

SEE ALSO

lp(1), lp(7)

[This page left blank.]

NAME

lpstat - print information about status of LP print service

SYNOPSIS

lpstat [options]

DESCRIPTION

The *lpstat* command prints information about the current status of the LP print service.

If no options are given, then *lpstat* prints the status of all requests made to *lp*(1) by the users. Any arguments that are not options are assumed to be *request-ids* (as returned by *lp*), printers, or printer classes. The *lpstat* command prints the status of such requests, printers, or printer classes. Options may appear in any order and may be repeated and intermixed with other arguments. Some of the keyletters in the following text may be followed by an optional *list* that can be in one of two forms; a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

```
-u "user1, user2, user3"
```

The omission of a list following such keyletters causes all information relevant to the keyletter to be printed. For example, the command

```
lpstat -o
```

prints the status of all output requests.

-a [*list*] Print acceptance status (with respect to *lp*) of destinations for requests [see *accept*(1M)]. The *list* is a list of intermixed printer names and class names.

-c [*list*] Print class names and their members. The *list* is a list of class names.

-d Print the system default destination for *lp*.

-o [*list*] [**-l**]
Print the status of output requests. The *list* is a list of intermixed printer names, class names, and *request-ids*.

-p [*list*] [**-D**] [**-l**]
Print the status of printers named in *list*.

-r Print the status of the LP request scheduler.

-s Print a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, a list of printers and their associated devices, a list of all forms currently mounted, and a list of all recognized character sets and print wheels.

-t Print all status information.

-u [*list*] Print status of output requests for users. *List* is a list of login names.

-v [*list*] Print the names of printers and the pathnames of the devices associated with them. *List* is a list of printer names.

Internationalization

Status messages containing characters from supplementary code sets can be displayed.

FILES

/usr/spool/lp/*

SEE ALSO

enable(1), lp(1)

NAME

ls, lc - list contents of directory

SYNOPSIS

ls [-RadCxmlnogrtucpFbqisf] [*names*]
lc [-Radxmlnogrtucpfbqisf] [*names*]

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.

The *lc* command is a synonym for *ls* with the -C option set. There are three major listing formats. The default format is to list one entry per line, the **-C** and **-x** options enable multi-column formats, and the **-m** option enables stream output format. In order to determine output formats for the **-C**, **-x**, and **-m** options, *ls* uses an environment variable, **COLUMNS**, to determine the number of character positions available on one output line. If this variable is not set, the *terminfo*(4) data base is used to determine the number of columns, based on the environment variable **TERM**. If this information cannot be obtained, 80 columns are assumed.

The *ls* command has the following options:

-R Recursively list subdirectories encountered.

-a List all entries, including those that begin with a dot (.), which are normally not listed.

-d If an argument is a directory, list only its name (not its contents); often used with **-l** to get the status of a directory.

-C Multi-column output with entries sorted down the columns (default setting for *lc*).

-x Multi-column output with entries sorted across rather than down the page.

-m Stream output format; files are listed across the page, separated by commas.

-l List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see the following text). If the file is a special file, the size field will instead contain the major and minor device numbers rather than a size.

-n The same as **-l**, except that the owner's UID and group's GID numbers are printed, rather than the associated character strings.

-o The same as **-l**, except that the group is not printed.

-g The same as **-l**, except that the owner is not printed.

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-t Sort by time stamp (latest first) instead of by name. The default is the last modification time. (See **-n** and **-c**.)

-u Use time of last access instead of last modification for sorting (with the **-t** option) or printing (with the **-l** option).

-c Use time of last modification of the inode (file created, mode changed, etc.) for sorting (**-t**) or printing (**-l**).

-p Put a slash (/) after each filename if that file is a directory.

-F Put a slash (/) after each filename if that file is a directory and put an asterisk (*) after each filename if that file is executable.

-b Force printing of non-printable characters to be in the octal **\ddd** notation.

-q Force printing of non-printable characters in filenames as the character question mark (?).

-i For each file, print the i-number in the first column of the report.

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

-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.

The mode printed under the **-l** option consists of ten characters. The first character may be one of the following.

d The entry is a directory.
b The entry is a block special file.
c The entry is a character special file.
m The entry is a XENIX shred data (memory) file.
p The entry is a fifo (a.k.a. "named pipe") special file.
s The entry is a XENIX semaphore.
- The entry is an ordinary file.

The next 9 characters are interpreted as three sets of three bits each. The first

set refers to the owner's permissions; the next to permissions of others in the user-group of the file; and the last to all others. Within each set, the three characters indicate permission to read, to write, and to execute the file as a program, respectively. For a directory, "execute" permission is interpreted to mean permission to search the directory for a specified file.

ls -l (the long list) prints its output as follows:

```
-rwxrwxrwx  1 smith   dev     10876  May 16 9:42 part2
```

This horizontal configuration provides a good deal of information. Reading from right to left, you see that the current directory holds one file, named "part2." Next, the last time that file's contents were modified was 9:42 A.M. on May 16. The file is moderately sized, containing 10,876 characters, or bytes. The owner of the file, or the user, belongs to the group "dev" (perhaps indicating "development"), and his or her login name is "smith." The number, in this case "1," indicates the number of links to file "part2." Finally, the row of dash and letters tell you that user, group, and others have permissions to read, write, execute "part2."

The execute (**x**) symbol here occupies the third position of the three-character sequence. A **-** in the third position would have indicated a denial of execution permissions.

The permissions are indicated as follows:

- **r** The file is readable.
- **w** The file is writable.
- **x** The file is executable.
- **-** The indicated permission is not granted.
- **l** Mandatory locking will occur during access (the set-group-ID bit is on and the group execution bit is off).
- **s** The set-user-ID or set-group-ID bit is on, and the corresponding user or group execution bit is also on.
- **S** Undefined bit-state (the set-user-ID bit is on and the user execution bit is off).
- **t** The 1000 (octal) bit, or sticky bit, is on [see *chmod*(1)], and execution is on.
- **T** The 1000 bit is turned on, and execution is off (undefined bit-state).

For user and group permissions, the third position is sometimes occupied by a character other than **x** or **-**. **s** also may occupy this position, referring to the state of the set-ID bit, whether it be the user's or the group's. The ability to assume

the same ID as the user during execution is, for example, used during login when you begin as root but need to assume the identity of the user stated at "login."

In the case of the sequence of group permissions, **l** may occupy the third position. The **l** refers to mandatory file and record locking. This permission describes a file's ability to allow other files to lock its reading or writing permissions during access.

For others permissions, the third position may be occupied by **t** or **T**. These refer to the state of the sticky bit and execution permissions.

Internationalization

The *ls* command can process directory and filenames containing characters from supplementary code sets. Multi-column output can be displayed correctly using the **-C** and **-x** options.

With the **-b** and **-q** options, *ls* considers all multibyte characters to be printable.

EXAMPLES

An example of a file's permissions is:

```
-rwxr--r--
```

This describes a file that is readable, writable, and executable by the user and readable by the group and others.

Another example of a file's permissions is:

```
-rwsr-xr-x
```

This describes a file that is readable, writable, and executable by the user, readable and executable by the group and others, and allows its user ID to be assumed, during execution, by the user presently executing it.

Another example of a file's permissions is:

```
-rw-rwl---
```

This describes a file that is readable and writable only by the user and the group and can be locked during access.

An example of a command line:

```
ls -a
```

This command will print the names of all files in the current directory, including those that begin with a dot (.), which normally do not print.

Another example of a command line:

ls -aisn

This command will provide you with quite a bit of information listing all files, including non-printing ones, with the **-a** option; printing the i-number (the memory address of the inode associated with the file) in the left-hand column with the **-i** option; giving the size (in blocks) of the files, printed in the column to the right of the i-numbers, with the **-s** option; and displaying the report in the numeric version of the long list, printing the **UID** (instead of user name) and **GID** (instead of group name) numbers associated with the files with the **-n** option.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks, is printed.

FILES

/etc/passwd	user IDs for **ls -l** and **ls -o**
/etc/group	group IDs for **ls -l** and **ls -g**
/usr/lib/terminfo/?/*	terminal information database

NOTES

In a Remote File Sharing environment, you may not have the permissions that the output of the **ls -l** command leads you to believe. For more information, see the Administration Guide.

SEE ALSO

chmod(1), find(1)

BUGS

Unprintable characters in filenames may confuse the columnar output options.

[This page left blank.]

NOTES

NOTES

NOTES

Document Update Notice
U 6000 Series System V

Title: System V User's Reference Manual, Volume 1, UP-15525 V1, Update A

Date: October 1989

Attached are update pages (marked UP-15525 V1 A) for the System V User's Reference Manual, Volume 1. Together with the title page, these pages upgrade the document to include the most recent information on the Unisys International Enhancements capability.

Remove	Insert
Title Page	Title Page
Table of Contents (pp. iii - ix)	Table of Contents (pp. iii - ix)
Permuted Index (pp. xi - xxi)	Permuted Index (pp. xi - xxi)
	gencat(1) (pp. 1-2)
	isastream(1) (pp. 1-2)
	keepopen(1) (pp. 1-2)

File this notice in the front of the manual to provide a record of changes.