PWB/UNIX

User's Manual

Edition 1.0

T. A. Dolotta
R. C. Haight
E. M. Piskorik

Editors

May 1977

Bell Telephone Laboratories, Incorporated

UNIX is a Trade/Service Mark of the Bell System.

This manual was set on a Graphic Systems, Inc. phototypesetter driven by the TROFF formatting program operating under the PWB/UNIX system. The text of the manual was prepared using the ED text editor.

ACKNOWLEDGEMENTS

The form and organization of this manual, as well as a major fraction of its contents, have been copied from the *UNIX Programmer's Manual—Sixth Edition*, by K. Thompson and D. M. Ritchie (Bell Telephone Laboratories, May 1975). The number of our colleagues who have contributed to UNIX and PWB/UNIX software and documentation is, by now, too large to list here, but the usefulness and acceptance of UNIX and of PWB/UNIX is a true measure of their collective success.

Piscataway, New Jersey T.A.D.
May 1977 R.C.H.
E.M.P.

INTRODUCTION

This manual describes the features of PWB/UNIX. It provides neither a general overview of UNIX (for that, see "The UNIX Time-Sharing System," *Comm. ACM* 17(7):365-75, July 1974, by D. M. Ritchie and K. Thompson), nor details of the implementation of the system.

This manual is divided into eight sections:

I. Commands and Application Programs
II. System Calls
III. Subroutines
IV. Special Files
V. File Formats and Conventions
VI. Games
VII. Miscellaneous
VIII. System Maintenance

Section I (*Commands and Application Programs*) describes programs intended to be invoked directly by the user or by command language procedures, in contradistinction to subroutines, which are intended to be called by the user's programs. Commands generally reside in the directory */bin* (for *bin*ary programs). Some programs also reside in */usr/bin*, to save space in */bin*. These directories are searched automatically by the command interpreter called the Shell.

Section II (*System Calls*) describes the entries into the UNIX supervisor, including the assembler and the C language interfaces. In the assembler, these system calls are invoked by the *sys* operation code, which is a synonym for the *trap* instruction.

Section III (*Subroutines*) describes the available subroutines. Their binary versions reside in various system libraries in directory */lib*. The subroutines available for C and for Fortran are also included there; they reside in */lib/libc.a* and */lib/libf.a*, respectively.

Section IV (*Special Files*) discusses the characteristics of each system "file" that actually refers to an input/output device. The names in that section refer to the Digital Equipment Corporation's device names for the hardware, instead of the names of the special files themselves.

Section V (*File Formats and Conventions*) documents the structure of particular kinds of files; for example, the form of the output of the assembler and the loader is given. Excluded are files used by only one command, for example, the assembler's intermediate files.

Section VIII (*System Maintenance*) discusses commands that are not intended for use by the ordinary user, in some cases because they disclose information in which he or she is presumably not interested, and in others because they perform privileged functions.

Each section consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages. Entries within each section are alphabetized. The page numbers of each entry start at 1.

All entries are based on a common format, not all of whose parts always appear:

The NAME part repeats the name of the entry and states (very briefly) its purpose.

The SYNOPSIS part summarizes the use of the program being described. A few conventions are used, particularly in Section I (*Commands*):

Boldface strings are considered literals, and are to be typed just as they appear (they are usually underlined in the typed version of the manual entries *unless* they are juxtaposed with an *italic* string).

Italic strings usually represent substitutable arguments (they are underlined in the typed version of the manual entries).

Square brackets "[]" around an argument indicate that the argument is optional. When an argument 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 sign "−" or a plus 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 "+".

The DESCRIPTION part discusses in detail the subject at hand.

The FILES part gives the file names that are built into the program.

The SEE ALSO part gives pointers to related information.

The DIAGNOSTICS part discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.

The BUGS part gives known bugs and sometimes deficiencies. Occasionally, the suggested fix is also described.

A table of contents (organized by section and alphabetized within each section) and a permuted index derived from that table precede Section I. Within each *index* entry, the title of the *manual* entry to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands that exist only to exercise a particular system call.

All manual entries are available on-line via the *man*(I) command (q.v.).

HOW TO GET STARTED

This section provides the basic information you need to get started on UNIX (we will use "UNIX" in this section to mean both "UNIX" and "PWB/UNIX", unless the distinction matters): how to log in and log out, how to communicate through your terminal, and how to run a program. See *UNIX for Beginners* by B. W. Kernighan for a more complete introduction to the system.

Logging in. You must call UNIX from an appropriate terminal. UNIX supports full-duplex ASCII terminals. You must also have a valid user name, which may be obtained, together with the telephone number, from the system administrator. The same telephone number serves terminals operating at speeds of 110, 150, and 300 baud. After a data connection is established, the log in procedure depends on the kind of terminal you are using.

> *300-baud terminals:* These terminals generally have a speed switch that should be set to "300" (or "30", for 30 characters per second) and a half-/full-duplex switch that should be set to full-duplex. When a connection is established, the system types "login:"; you type your user name, followed by the "return" key. If you have a password (and you should!), the system asks for it, but does not print ("echo") it on the terminal. After you have logged in, the "return", "new-line", and "line-feed" keys will give exactly the same result.
>
> *Model 37 TELETYPE®:* When you have established a data connection, the system types out a few garbage characters (the "login:" message at the wrong speed). Depress the "break" (or "interrupt") key; this is a speed-independent signal to UNIX that a 150-baud terminal is in use. The system then will type "login:", this time at 150 baud (another "break" at this point will get you down to 110 baud); you respond with your user name. From the Model 37 TELETYPE, and any other terminal that has the "new-line" function (combined "carriage-return" and "line-feed" pair), terminate each line you type with the "new-line" key (*not* the "return" key).

It is important that you type your name in lower case if possible; if you type upper-case letters, UNIX will assume that your terminal cannot generate lower-case letters and will translate all subsequent upper-case input to lower case. When you have logged in successfully, the Shell program will type a "%" to you. (The Shell is described below under *How to run a program.*)

For more information, consult *login*(I) and *getty*(VIII), which discuss the login sequence in more detail, and *tty*(IV), which discusses terminal input/output. See *terminals*(VII) for information about various terminals.

Logging out. There are three ways to log out:

> You can simply hang up the phone.
>
> You can log out by typing an end-of-file indication (ASCII EOT character, usually typed as "control d") to the Shell. The Shell will terminate and the "login:" message will appear again.
>
> You can also log in directly as another user by giving a *login* command.

How to communicate through your terminal. When you type to UNIX, a gnome deep in the system is gathering your characters and saving them. These characters will not be given to a program until you type a "return" (or "new-line"), as described above in *Logging in.*

UNIX 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 typing at you. Of course, if you type during output, the output will have the input characters interspersed. However, whatever you type will be saved and interpreted in correct sequence. There is a limit to the amount of read-ahead, but it

is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away *all* the saved characters.

On a terminal input line, the character "@" kills all the characters typed before it, so typing mistakes can be repaired on a single line. The character "#" erases the last character typed. Successive uses of "#" erase characters back to, but not beyond, the beginning of the line. "@" and "#" can be transmitted to a program by preceding them with "\". (Thus, to erase "\", you need two "#"s).

The ASCII "delete" (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 don't want. However, programs can arrange either to ignore this signal altogether, or to be notified when it happens (instead of being terminated). The editor *ed*(I), for example, catches interrupts and stops what it is doing, instead of terminating, so that an interrupt can be used to halt an editor printout without losing the file being edited.

The *quit* signal is generated by typing the ASCII FS character. It not only causes a running program to terminate but also generates a file with the core image of the terminated process. *Quit* is useful for debugging.

Besides adapting to the speed of the terminal, UNIX tries to be intelligent about 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*(I) command will rescue you.

Tab characters are used freely in UNIX 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*(I) command will set or reset this mode. The system assumes that tabs are set every eight columns. The *tabs*(I) command will set tab stops on your terminal, if that is possible.

How to run a program; the Shell. When you have successfully logged into UNIX, a program called the Shell is listening to your terminal. The Shell reads typed-in lines, splits them up into a command name and 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* below) for a program with the given name, and if none is there, then in system directories. There is nothing special about system-provided commands except that they are kept in directories where the Shell can find them. The command name is always the first word on an input line to the Shell; it and its arguments are separated from one another by space or tab characters.

When a program terminates, the Shell will ordinarily regain control and type a "%" at you to indicate that it is ready for another command. The Shell has many other capabilities, which are described in detail in *sh*(I).

The current directory. UNIX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he or she also created a directory for you (ordinarily with the same name as your user name). When you log in, any file name you type is by default assumed to be in this directory. Since you are the owner of this directory, you have full permissions to read, write, alter, or destroy its contents. Permissions to have your will with other directories and files will have been granted or denied to you by their respective owners. As a matter of observed fact, many UNIX users do not protect their files from destruction, let alone perusal, by other users.

To change the current directory (but not the set of permissions you were endowed with at login) use *chdir*(I).

Path names. To refer to files not in the current directory, you must use a path name. Full path names 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 name is reached. E.g.: */usr/lem/filex* refers to the file *filex* in the directory *lem; lem* is itself a subdirectory of *usr; usr* springs directly from the root directory.

If your current directory has subdirectories, the path names of files therein begin with the name of the corresponding subdirectory (without a prefixed "/").

Without important exception, a path name may be used anywhere a file name is required.

Important commands that modify the contents of files are *cp*(I), *mv*(I), and *rm*(I), which respectively copy, move (i.e., rename), and remove files. To find out the status of files or directories, use *ls*(I). See *mkdir*(I) for making directories and *rmdir*(I) for destroying them.

For a fuller discussion of the file system, see "The UNIX Time-Sharing System" (*Comm. ACM* 17(7):365-75, July 1974) by D. M. Ritchie and K. Thompson. It may also be useful to glance through Section II of this manual, which discusses system calls, even if you don't intend to deal with the system at that level.

Writing a program. To enter the text of a source program into a UNIX file, use *ed*(I). The three principal languages available under UNIX are C (see *cc*(I)), Fortran (see *fc*(I)), and assembly language (see *as*(I)). After the program text has been entered through the editor and written in a file (whose name has the appropriate suffix), you can give the name of that file to the appropriate language processor as an argument. Normally, the output of the language processor will be left in a file in the current directory named "a.out". (If the output is precious, use *mv*(I) to move it to a less exposed name soon.) If you wrote in assembly language, you will probably need to load the program with library subroutines; see *ld*(I). The other two language processors call the loader automatically.

When you have finally gone through this entire process without provoking any diagnostics, the resulting program can be run by giving its name to the Shell in response to the "%" prompt.

Next, you will need *cdb*(I) or *db*(I) to examine the remains of your program. The former is useful for C programs, the latter for assembly-language. No debugger is much help for Fortran.

Your programs can receive arguments from the command line just as system programs do. See *exec*(II).

Text processing. Almost all text is entered through the editor *ed*(I). The commands most often used to write text on a terminal are: *cat*(I), *pr*(I), and *nroff*(I). The *cat*(I) command simply dumps ASCII text on the terminal, with no processing at all. The *pr*(I) command paginates the text, supplies headings, and has a facility for multi-column output. *Nroff*(I) is an elaborate text formatting program, and requires careful forethought in entering both the text and the formatting commands into the input file; it produces output on a typewriter-like terminal. *Roff*(I) is a less elaborate text formatting program, and requires somewhat less forethought; it is obsolescent. *Troff*(I) is similar to *nroff*(I), but drives a Graphic Systems, Inc. phototypesetter. It was used to typeset this manual.

Surprises. 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 aim them at you. To communicate with another user currently logged in, *write*(I) is used; *mail*(I) will leave a message whose presence will be announced to another user when he or she next logs in. The corresponding entries in this manual also suggest how to respond to these two commands if you are their target.

When you log in, a message-of-the-day may greet you before the first "%".

TABLE OF CONTENTS

I. COMMANDS AND APPLICATION PROGRAMS

II. SYSTEM CALLS

III. SUBROUTINES

IV. SPECIAL FILES

V. FILE FORMATS AND CONVENTIONS

VI. GAMES

VII. MISCELLANEOUS

VIII. SYSTEM MAINTENANCE

PERMUTED INDEX

NAME

450 – handle special functions of DASI450 terminal

SYNOPSIS

450

DESCRIPTION

450 supports special functions of, and optimizes the use of the DASI450 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 *gsi(I)*. *450* can be used to print equations neatly, in the sequence:

neqn file ... | nroff | 450

NOTE: *450* 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 any response.

However, in most cases, *450* can be eliminated in favor of the following:

nroff –T450 *files...* or **nroff –T450–12** *files...*

In a few cases, the additional movement optimization of *450* may produce better-aligned output.

The SPACING switch may be in either 10-pitch or 12-pitch position (but that setting can be overridden dynamically). In either case, vertical spacing is 6 lines/inch, unless dynamically changed to 8 lines per inch by an appropriate escape sequence.

SEE ALSO

graph(I), gsi(I), mesg(I), neqn(I), stty(I), tabs(I), greek(V), DASI450(VII), terminals(VII)

BUGS

Some Greek characters can't be correctly printed in column 1 because the print head cannot be moved to the left from there. If your output contains much Greek and/or reverse line feeds, use friction feed instead of a forms tractor. Although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters, and misaligning the first line after a long set of reverse line feeds.

NAME

adb – debugger

SYNOPSIS

adb [**–w**] [objfil [corfil]]

DESCRIPTION

Adb is a general-purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of UNIX programs.

Objfil is normally an executable program file, preferably containing a symbol table; if not then the symbolic features of *adb* cannot be used although the file can still be examined. The default for *objfil* is **a.out**. *Corfil* is assumed to be a core image file produced after executing *objfil;* the default for *corfil* is **core.**

Requests to *adb* are read from the standard input and responses are to the standard output. If the **–w** flag is present then both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using *adb*. *Adb* ignores QUIT signals; INTERRUPT causes return to the next *adb* command.

In general, requests to *adb* are of the form

[*address*] [, *count*] [*command*] [;]

If *address* is present then *dot* is set to *address*. Initially *dot* is set to 0. For most commands, *count* specifies how many times the command will be executed. The default *count* is 1; *Address* and *count* are expressions.

The interpretation of an address depends on the context it is used in. If a sub-process is being debugged then addresses are interpreted in the usual way in the address space of the sub-process. For further details of address mapping see **ADDRESSES.**

EXPRESSIONS

. The value of *dot.*

\+ The value of *dot* incremented by the current increment.

^ The value of *dot* decremented by the current increment.

" The last *address* typed.

integer An octal number if *integer* begins with a 0; a hexadecimal number if preceded by '#'; otherwise a decimal number.

integer.fraction
A 32-bit floating point number.

'cccc' The ASCII value of up to 4 characters. '\' may be used to escape '".

< name The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (q.v.) that are referred to by the letters **a** to **z** or the digits 0 to 9 (see **VARIABLES** below). If *name* is a register name, then the value of the register is obtained from the system header in *corfil.* The register names are **r0 ... r5 sp pc ps**.

symbol A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. '\' may be used to escape other characters. The value of the *symbol* is taken from the symbol table in *objfil.* An initial '_' or '~' will be prepended to *symbol* if needed.

routine.*name*
The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted, the value is the address of the most recently activated C stack frame corresponding to *routine*.

(*exp*) The value of *exp*.

Monadic operators

* *exp* The contents of the location addressed by *exp* in *corfil*.

@*exp* The contents of the location addressed by *exp* in *objfil*.

– *exp* Integer negation.

~ *exp* Bitwise complement.

Dyadic operators are left associative and are less binding than monadic operators.

e1 + *e2* Integer addition.

e1 – *e2* Integer subtraction.

e1 * *e2* Integer multiplication.

e1 % *e2* Integer division.

e1 & *e2* Bitwise conjunction.

e1 | *e2* Bitwise disjunction.

e1 # *e2* *e1* rounded up to the next multiple of *e2*.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands '?' and '/' may be followed by '*'; see ADDRESSES for further details.)

? *f* Locations starting at *address* in *objfil* are printed according to the format *f*. *Dot* is incremented by the sum of the increments for each format letter (q.v.).

/ *f* Locations starting at *address* in *corfil* are printed according to the format *f* and *dot* is incremented as for '?'.

= *f* The value of *address* itself is printed in the styles indicated by the format *f*. (For i format '?' is printed for the parts of the instruction that reference subsequent words.)

Formats

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format, *dot* is incremented by the amount given for each format letter. If no format is given then the last format is used. The format letters available are as follows.

o	2	Print 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O	4	Print 4 bytes in octal.
q	2	Print in signed octal.
Q	4	Print long signed octal.
d	2	Print in decimal.
D	4	Print long decimal.
x	2	Print 2 bytes in hexadecimal.

X	4	Print 4 bytes in hexadecimal.
u	2	Print as an unsigned decimal number.
U	4	Print long unsigned decimal.
f	4	Print the 32-bit value as a floating point number.
F	8	Print double floating point.
b	1	Print the addressed byte in octal.
c	1	Print the addressed character.
C	1	Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s	*n*	Print the addressed characters until a zero character is reached.
S	*n*	Print a string using the @ escape convention; *n* is the length of the string including its zero terminator.
Y	4	Print 4 bytes in date format (see *time*(II)).
i	*n*	Print as PDP-11 instructions; *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a	0	Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.
p	2	Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.
t	0	When preceded by an integer, tabs to the next appropriate tab stop. For example, **8t** moves to the next 8 space tab stop.
r	0	Print a space.
n	0	Print a newline.
"..."	0	Print the enclosed string.
^		*dot* is decremented by the current increment. Nothing is printed.
+		*dot* is incremented by 1. Nothing is printed.
−		*dot* is decremented by 1. Nothing is printed.

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

MORE COMMANDS

Here are a few more commands; '[?/]' means the command can start with either '?', for addresses in *objfil*, or '/', for addresses in *corfil*.

[?/] l *value mask*

Words starting at *dot* are masked with *mask* and compared with *value* until a match is found. If L is used, then the match is for 4 bytes at a time instead of 2. If no match is found, then *dot* is unchanged; otherwise *dot* is set to the matched location. If *mask* is omitted, then −1 is used.

[?/] **w** *value* ...

value is written into the addressed location. If **W** is used then 4 bytes are written, otherwise 2 bytes are written. Odd addresses are not allowed when writing to the sub-process address space.

[?/] **m** *b1 e1 f1* [?/]

New values for (*b1*, *e1*, *f1*) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second segment (*b2*, *e2*, *f2*) of the mapping is changed. If the list is terminated by '?' or '/' then the file (*objfil* or *corfil* respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to *objfil*.)

> *name* *dot* is assigned to the variable or register named.

! A shell is called to read the rest of the line following '!'.

$ *modifier*

< *f* Read commands from the file *f* and return.

> *f* Send output to the file *f* which is created if it does not exist.

r Print the general registers and the instruction addressed by **pc**; *dot* is set to **pc**.

f Print the floating registers in single or double length. If the floating point status of **ps** is set to double (0200 bit) then double length is used anyway.

b Print all breakpoints and their associated counts and commands.

a ALGOL 68 stack backtrace. If *address* is given then it is taken to be the address of the current frame (instead of **r4**). If *count* is given then only the first *count* frames are printed.

c C stack backtrace. If *address* is given then it is taken as the address of the current frame (instead of **r5**). If **C** is used then the names and (16-bit) values of all automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.

e The names and values of external variables are printed.

w Set the page width for output to *address* (default 80).

s Set the limit for symbol matches to *address* (default 255).

o All integers input are regarded as octal.

d Reset integer input as described in **EXPRESSIONS**.

q Exit from *adb*.

v Print all non-zero variables in octal.

m The values used for mapping addresses into file addresses are printed.

: *modifier*

b *c* Set breakpoint at *address*. The breakpoint is executed *c*−*1* times before causing a stop. Each time the breakpoint is encountered, the command *c* is executed. If this command sets *dot* to zero then the breakpoint causes a stop.

d Delete breakpoint at *address*.

r *c* Run *objfil* as a sub-process. If *address* is given explicitly, then the program is entered at this point; otherwise, the program is entered at its standard entry point; *c* specifies how many breakpoints are to be ignored before stopping. Arguments to the sub-process may be supplied on the same line as the command. An argument starting with < or > causes the standard input or

output to be established for the command. All signals are turned on on entry to the sub-process.

c *s* The sub-process is continued with signal *s*. If *address* is given then the sub-process is continued at this address. If no signal is specified then the signal that caused the sub-process to stop is sent. Breakpoint skipping is the same as for **r**.

s *s* As for **c** except that the sub-process is single stepped *count* times. If there is no current sub-process then *objfil* is run as a sub-process as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the sub-process.

k The current sub-process, if any, is terminated.

VARIABLES

Adb provides a number of variables. Named variables are set initially by *adb* but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

On entry the following are set from the system header in the *corfil*. If *corfil* does not appear to be a **core** file then these values are set from *objfil*.

b The base address of the data segment.
d The data segment size.
e The entry point.
m The 'magic' number (0405, 0407, 0410 or 0411).
s The stack segment size.
t The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1*, *e1*, *f1*) and (*b2*, *e2*, *f2*) and the *file address* corresponding to a written *address* is calculated as follows.

$$b1 \leqslant address < e1 \Rightarrow file\ address = address + f1 - b1, \text{ otherwise,}$$

$$b2 \leqslant address < e2 \Rightarrow file\ address = address + f2 - b2,$$

otherwise, the requested *address* is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a '?' or '/' is followed by an '*' then only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size and *f1* is set to 0; in this way the whole file can be examined with no address translation.

So that *adb* may be used on large files all appropriate values are kept as signed 32-bit integers.

EXIT STATUS

If the last command was successful then the exit status is zero; otherwise it is non-zero.

FILES

/dev/mem
/dev/swap

SEE ALSO

cdb(I), db(I), ptrace(II), a.out(V), core(V)

BUGS

a) A breakpoint set at the entry point is not effective on initial entry to the program.

b) When single stepping, system calls do not count as an executed instruction.

NAME

admin – administer SCCS files

SYNOPSIS

admin [**–n**] [**–i**[name] [**–r**rel]] [**–t**[name]] [**–f**add-flag[flag-val]] ... [**–d**delete-flag] ... [**–a**add-login] ... [**–e**erase-login] ... [**–h**] [**–z**] name ...

DESCRIPTION

Admin is used to create new SCCS files and change parameters of existing ones. Arguments to *admin*, which may appear in any order, consist of keyletter arguments, which begin with "–", and named files. If a named file doesn't exist, it is created, and its parameters are initialized according to the specified keyletter arguments. Parameters not initialized by a keyletter argument are assigned a default value. If a named file does exist, parameters corresponding to specified keyletter arguments are changed, and other parameters are left as is.

If a directory is named, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files, are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument other than **i** and **r** apply independently to each named file.

–n This argument indicates that new files are to be created. This argument must be specified when creating new SCCS files. The **i** argument implies an **n** argument.

–i The name of a file from which the text of an initial delta is to be taken. If this argument is supplied, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this argument is omitted, and the *admin* command creates one or more SCCS files, then their initial deltas must be inserted in the normal manner, using *get* and *delta*(I). Only one SCCS file may be created by an *admin* command on which the **i** argument is supplied.

–r The release into which the initial delta will be inserted. This argument may only be supplied if the **i** argument is also supplied. If this argument is omitted, the initial delta will be inserted into release 1. The level of the initial delta will always be 1.

–t The name of a file from which descriptive text for the SCCS file is to be taken. If this argument is supplied and *admin* is creating a new SCCS file, the descriptive-text filename must also be supplied. In the case of existing SCCS files, if this argument is supplied but the file name is omitted, the descriptive text (if any) currently in the SCCS file will be removed. If the file name is supplied, the text in the file named will replace the descriptive text (if any) currently in the SCCS file.

–f This argument specifies a flag, and, possibly, a value for the flag, to be added to the SCCS file. Several **f** arguments may be supplied on a single *admin* command. The allowable flags and their values are as follows:

b The presence of this flag indicates that the use of the **b** argument on a *get* command will cause a branch to be taken in the delta tree..

cceil The "ceiling:" the highest release (less than or equal to 9999) which may be specified by the **r** argument on a *get* with an **e** argument. If this flag is not specified, the ceiling is 9999.

dSID The default SID to be used on a *get* when the **r** argument is not supplied.

ffloor The "floor:" the lowest release (greater than 0) which may be specified by the **r** argument on a *get* with an **e** argument. If this flag is not specified, the floor is 1.

i The presence of this flag causes the "No id keywords (ge6)" message issued by *get* or *delta* to be treated as a fatal error. In the absence of this flag, the message is only a warning.

mmod This flag specifies the module name of the SCCS file. Its value will be used by *get* as the replacement for the %M% keyword.

ttype This flag specifies the type of the module. Its value will be used by *get* as a replacement for the %Y% keyword.

v[pgm] The presence of this flag indicates that *delta* is to prompt for MR numbers in addition to comments. If the optional value of this flag is present, it specifies the name of an MR number validity checking program.

—d This argument specifies a flag to be completely removed from an SCCS file. This argument may only be specified when processing existing SCCS files. Several **d** arguments may be supplied on a single *admin* command. See the **f** argument for the allowable flags.

—a A login name to be added to the list of logins which may add deltas. Several **a** arguments may be supplied on a single *admin* command. As many logins as desired may be on the list simultaneously. If the list of logins is empty, then anyone may add deltas.

—e A login name to be erased from the list of logins. Several **e** arguments may be supplied on a single *admin* command.

—h This argument provides a convenient mechanism for checking for corrupted files. With this argument, *admin* will check that the sum of all the characters in the SCCS file (the check-sum) agrees with the sum which is stored in the first line of the file. If the sums are not in agreement a "corrupted file" message will be produced. This argument inhibits writing on the file, so that it will nullify the effect of any other arguments supplied, and is, therefore, only meaningful when processing existing files.

—z This argument will cause *admin* to ignore any discrepancy in the check-sum of the SCCS file (see **h** argument), and to replace it with the new one. (The same effect may be had by first editing the SCCS file with *ed*(I) in order to replace the five-character check-sum in the first line of the file with five zeroes. A subsequent invocation of an SCCS command which modifies the file (e.g., *admin*, *delta*), will cause check-sum validation to be by-passed, and a new check-sum to be computed.) The purpose of this is

to correct the check-sum in those files which may have been edited by the user. Note that use of this argument on a truly corrupted file will prevent future detection of the corruption.

FILES

The last component of all SCCS file names must be of the form "s.*modulename*". New SCCS files are given mode 444. Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file (see *get*(I)), created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file will be deleted, if it exists, and the x-file will be renamed with the name of the SCCS file. This ensures that changes will be made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be mode 444. The mode of the directories will allow only the owner to modify SCCS files contained in the directories. The mode of the SCCS files will prevent any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner, and then the owner may edit the file at will with *ed*(I).

Admin also makes use of the *z*-file, which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(I) for further information.

SEE ALSO

get(I), delta(I), prt(I), what(I), help(I), ed(I), sccsfile(V)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

ar – archive and library maintainer

SYNOPSIS

ar key [posname] afile name ...

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the *ld*(I). It can be used, though, for any similar purpose.

Key is one character from the set **drtpmx**, optionally concatenated with **vuabin**. *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d means delete the named files from the archive file.

r means replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with modified dates later than the archive files are replaced. If the optional positioning character **a** (also **i** or **b**) is used, then the *posname* argument must be present and specifies a file in the archive after (before for **i** and **b**) which new files are placed. Without **a**, **i**, or **b**, new files are placed at the end.

t prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p prints the named files in the archive.

m moves the named files to the end of the archive. If the options **a**, **i**, or **b** are used, then the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.

x extracts the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v means verbose. Under the verbose option, *ar* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, it gives a long listing of all information about the files.

n is accepted with no effect whatsoever.

In all cases, the archive file is created mode 644.

FILES

/tmp/v????? temporary
/tmp/v1????? temporary
/tmp/v2????? temporary

DIAGNOSTICS

Most diagnostics are self-explanatory. The message "no space in *xxx*" means that the file system *xxx* does not have enough space to contain the temporary files or the new archive file.

SEE ALSO

ld(I), archive(V)

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

as – assembler

SYNOPSIS

as [–] [–**o** objfil] name...

DESCRIPTION

As assembles the concatenation of the named files. If the optional first argument – is used, all undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file *objfil;* if that is omitted, **a.out** is used. It is executable if no errors occurred during the assembly, and if there were no unresolved external references.

FILES

/lib/as2	pass 2 of the assembler
/tmp/atm[1-3]?	temporary
a.out	object

SEE ALSO

ld(I), nm(I), db(I), a.out(V), *UNIX Assembler Reference Manual* by D. M. Ritchie.

DIAGNOSTICS

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	Parentheses error
]	Parentheses error
<	String not terminated properly
*	Indirection used illegally
.	Illegal assignment to '.'
A	Error in address
B	Branch instruction is odd or too remote
E	Error in expression
F	Error in local ('f' or 'b') type symbol
G	Garbage (unknown) character
I	End of file inside an if
M	Multiply defined symbol as label
O	Word quantity assembled at odd address
P	'.' different in pass 1 and 2
R	Relocation error
U	Undefined symbol
X	Syntax error

BUGS

Symbol table overflow is not checked. **x** errors can cause incorrect line numbers in following diagnostics.

NAME

banner – print in block letters

SYNOPSIS

banner arg ...

DESCRIPTION

Banner writes characters as large block letters, 7 characters by 7 characters, on the standard output file. Each argument may be up to ten characters, and is printed on a separate row.

NAME

bas – basic

SYNOPSIS

bas [file]

DESCRIPTION

Bas is a dialect of Basic. If a file argument is provided, the file is used for input before the console is read. *Bas* accepts lines of the form:

statement
integer statement

Integer numbered statements (known as internal statements) are stored for later execution. They are stored in sorted ascending order. Non-numbered statements are immediately executed. The result of an immediate expression statement (that does not have '=' as its highest operator) is printed.

Statements have the following syntax:

expression
The expression is executed for its side effects (assignment or function call) or for printing as described above.

comment ...
This statement is ignored. It is used to interject commentary in a program.

done
Return to system level.

draw expression expression expression
A line is drawn on the Tektronix 611 display '/dev/vt0' from the current display position to the XY co-ordinates specified by the first two expressions. The scale is zero to one in both X and Y directions. If the third expression is zero, the line is invisible. The current display position is set to the end point.

display list
The list of expressions and strings is concatenated and displayed (i.e. printed) on the 611 starting at the current display position. The current display position is not changed.

dump
The name and current value of every variable is printed.

edit
The UNIX editor, *ed*, is invoked with the *file* argument. After the editor exits, this file is recompiled.

erase
The 611 screen is erased.

for name = expression expression statement
for name = expression expression
...
next
The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the

second expression.

goto expression
The expression is evaluated, truncated to an integer and execution goes to the corresponding integer numbered statment. If executed from immediate mode, the internal statements are compiled first.

if expression statement
if expression
...
[**else**
...]
fi
The statement (first form) or group of statements (second form) is executed if the expression evaluates to non-zero. In the second form, an optional **else** allows for a group of statements to be executed when the first group is not.

list [expression [expression]]
is used to print out the stored internal statements. If no arguments are given, all internal statements are printed. If one argument is given, only that internal statement is listed. If two arguments are given, all internal statements inclusively between the arguments are printed.

print list
The list of expressions and strings are concatenated and printed. (A string is delimited by " characters.)

prompt list
Prompt is the same as *print* except that no newline character is printed.

return [expression]
The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

run
The internal statements are compiled. The symbol table is re-initialized. The random number generator is reset. Control is passed to the lowest numbered internal statement.

save [expression [expression]]
Save is like *list* except that the output is written on the *file* argument. If no argument is given on the command, **b.out** is used.

Expressions have the following syntax:

name
A name is used to specify a variable. Names are composed of a letter followed by letters and digits. The first four characters of a name are significant.

number
A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an e followed by a possibly signed exponent.

(expression)
Parentheses are used to alter normal order of evaluation.

_ expression
The result is the negation of the expression.

expression operator expression

Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. A complete list of operators is given below.

expression ([expression [, expression] ...])

Functions of an arbitrary number of arguments can be called by an expression followed by the arguments in parentheses separated by commas. The expression evaluates to the line number of the entry of the function in the internally stored statements. This causes the internal statements to be compiled. If the expression evaluates negative, a builtin function is called. The list of builtin functions appears below.

name [expression [, expression] ...]

Each expression is truncated to an integer and used as a specifier for the name. The result is syntactically identical to a name. **a[1,2]** is the same as **a[1][2]**. The truncated expressions are restricted to values between 0 and 32767.

The following is the list of operators:

=

= is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

& |

& (logical and) has result zero if either of its arguments are zero. It has result one if both its arguments are non-zero. | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments are non-zero.

< <= > >= == <>

The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, == equal to, <> not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: a>b>c is the same as a>b&b>c.

+ −

Add and subtract.

* /

Multiply and divide.

^

Exponentiation.

The following is a list of builtin functions:

arg(i)

is the value of the *i*-th actual parameter on the current level of function call.

exp(x)

is the exponential function of *x*.

log(x)

is the natural logarithm of *x*.

sqr(x)

is the square root of *x*.

sin(x)

is the sine of *x* (radians).

cos(x)
is the cosine of x (radians).

atn(x)
is the arctangent of x. Its value is between $-\pi/2$ and $\pi/2$.

rnd( )
is a uniformly distributed random number between zero and one.

expr( )
is the only form of program input. A line is read from the input and evaluated as an expression. The resultant value is returned.

abs(x)
is the absolute value of x.

int(x)
returns x truncated (towards 0) to an integer.

FILES

/tmp/btm?	temporary
b.out	save file

DIAGNOSTICS

Syntax errors cause the incorrect line to be typed with an underscore where the parse failed. All other diagnostics are self explanatory.

BUGS

Has been known to give core images.

NAME

bc – arbitrary precision interactive language

SYNOPSIS

bc [**–l**] [file ...]

DESCRIPTION

Bc is an interactive processor for a language which resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The '–l' argument stands for the name of a library of mathematical subroutines which contains sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), and exponential ('e'). The syntax for *bc* programs is as follows; E means expression, S means statement.

Comments

are enclosed in /* and */.

Names

letters a–z
array elements: letter[E]
The words 'ibase', 'obase', and 'scale'

Other operands

arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
<letter> (E , ... , E)

Operators

+ – * / % ^
++ –– (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 are exemplified by

```
define <letter> ( <letter> ,..., <letter> ) {
        auto <letter>, ... , <letter>
        S; ... S
        return ( E )
}
```

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. Assignments to *ibase* or *obase* set

the input and output number radix respectively.

The same letter may be used as an array name, a function name, and a simple variable simultaneously. 'Auto' variables are saved and restored during function calls. All other variables are global to the program. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

For example

```
scale = 20
define e(x){
        auto a, b, c, i, s
        a = 1
        b = 1
        s = 1
        for(i=1; 1==1; i++){
                a = a*x
                b = b*i
                c = a/b
                if(c == 0) return(s)
                s = s+c
        }
}
```

defines a function to compute an approximate value of the exponential function and

```
        for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

FILES

/usr/lib/lib.b mathematical library

SEE ALSO

dc(1)
C Reference Manual by D. M. Ritchie.
BC — An Arbitrary Precision Desk Calculator Language by L. L. Cherry and R. Morris.

BUGS

No &&, || yet.
for statement must have all three E's
quit is interpreted when read, not when executed.

NAME

bdiff – big diff

SYNOPSIS

bdiff name1 name2

DESCRIPTION

Bdiff is used in a manner analogous to *diff*(I) to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for *diff*(I). *Bdiff* splits the files into 4000-line segments, and invokes *diff*(I) on corresponding segments (4000-lines is a reasonable upper limit for *diff(I)*). If *name1* (*name2*) is "–", the standard input is read. The output of *bdiff* is exactly that of *diff*(I), with line numbers adjusted to account for the segmenting (that is, to make it look as if the files had been processed whole).

Note that unlike *diff*(I), *bdiff* supports no optional keyletter arguments. In addition, 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(I)

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

bfs – big file scanner

SYNOPSIS

bfs [–] name

DESCRIPTION

Bfs is (almost) like *ed*(I) except that it is read-only and processes much bigger files. Files can be up 1024K bytes (the maximum possible size) and 32K lines, with up to 255 characters per line. *Bfs* is usually more efficient than *ed* for scanning a file, since the file is not copied to a buffer.

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 is typed as in *ed*. 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* are supported. In addition, regular expressions may be surrounded with two symbols besides '/' and '?': '>' indicates downward search without wrap-around, and '<' indicates upward search without wrap-around. Since *bfs* uses a different regular expression-matching routine from *ed*, the regular expressions accepted are slightly wider in scope (see *regex*(III)). 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, n, p, q, w, =, !* and null commands operate as described under *ed*. 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* file name. The *w* command is independent of output diversion, truncation or crunching (see the *xo, xt* and *xc* commands, below). 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*. *Xf* commands may be nested to a depth of 10.

xo [file]
: Further output from the *p* and null commands is diverted to the named file, which, if necessary, is created mode 666. Plain 'xo' diverts output back 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 the named 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 doesn't match at least one line in the specified range, including the first and last lines.

On success, '.' is set to the line matched and a jump is made to the label. This command is the only one that doesn't 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: 0–9][optional spaces][value]

The variable name is the specified *digit* following the 'xv'. 'xv5100' or 'xv5 100' both assign the *value* '100' to the *variable* '5'. 'xv61,100p' assigns the *value* '1,100p' to *variable* '6'. To reference the variable put a '%' in front of the variable name. For example, using the above assignments for the 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 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

xbn label

These two commands will test the last saved *return code* from the execution of a unix command (!UNIX command) and branch on a zero or nonzero 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 isn't. Plain 'xc' reverses the switch. Initially the switch is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

SEE ALSO

ed(I), regex(III)

DIAGNOSTICS

'?' for errors in commands, if prompting is turned off. Self-explanatory error messages when prompting is on.

NAME

cal – print calendar

SYNOPSIS

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

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

NAME

cat – concatenate and print

SYNOPSIS

cat [**–s**] [**–u**] file ...

DESCRIPTION

Cat reads each *file* in sequence and writes it on the standard output. Thus

cat file

prints the file, and

cat file1 file2 >file3

concatenates the first two files and places the result on the third.

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

The **–s** flag suppresses the error messages that *cat* would otherwise give for non-existent (or unreadable) files. The **–u** flag causes *cat* to work in an unbuffered fashion (read one character, then write that character).

SEE ALSO

pr(I), cp(I)

DIAGNOSTICS

file not found

BUGS

cat x y >x and **cat x y >y** cause strange results (because of *sh*(I)).

NAME

cb – C beautifier

SYNOPSIS

cb

DESCRIPTION

cb reads a C program from the standard input, adds the proper indentation, and writes it on the standard output.

NAME

cc – C compiler

SYNOPSIS

cc [−c] [−p] [−f] [−Dn=v] [−Idir] [−O] [−S] [−P] [−Un] files ...

DESCRIPTION

Cc is the UNIX C compiler. It accepts three types of arguments:

Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

The following flags are interpreted by *cc*. See *ld*(I) for load-time flags.

−c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

−p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls the *monitor*(III) subroutine at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(I).

−f In systems without hardware floating-point, use a version of the C compiler which handles floating-point constants and loads the object program with the floating-point interpreter. Do not use if the hardware is present.

−D The name *n* is *defined*, and is given the value *v*, if specified.

−O Invoke an object-code optimizer.

−S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

−P Run only the macro preprocessor on the named C programs, and leave the output on corresponding files suffixed '.i'.

−U The name *n* is *undefined*.

−I The *include* preprocessor statement looks in directory *dir* if it can't find the specified file in the local directory or in **/usr/include.**

Other arguments are taken to be either loader flag arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.** If desired, a different name can be used; see the **−o** option of *ld*(I).

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/c[01]	compiler
/lib/fc[01]	floating-point compiler

/lib/c2	optional optimizer
/lib/cpp	pre-processor
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	runtime startoff of profiling
/lib/fcrt0.o	runtime startoff for floating-point interpretation
/lib/libc.a	C library; see section III.
/lib/liba.a	Assembler library used by some routines in libc.a

SEE ALSO

C Reference Manual by D. M. Ritchie.
Programming in C — A Tutorial by B. W. Kernighan.
adb(I), cdb(I), ld(I), prof(I), monitor(III)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader (see *as*(I) and *ld*(I)). Of these, the most mystifying are from the assembler, in particular "m," which means a multiply-defined external symbol (function or data).

NAME

cd – change working directory

SYNOPSIS

cd directory

DESCRIPTION

Cd is an alias for *chdir*(I).

SEE ALSO

chdir(I), sh(I), pwd(I)

NAME

cdb – C debugger

SYNOPSIS

cdb [a.out [core]]

DESCRIPTION

Cdb is a debugger for use with C programs. It is useful for both post-mortem and interactive debugging. An important feature of *cdb* is that even in the interactive case no advance planning is necessary to use it; in particular it is not necessary to compile or load the program in any special way nor to include any special routines in the object file.

The first argument to *cdb* is an object program, preferably containing a symbol table; if not given "a.out" is used. The second argument is the name of a core-image file; if it is not given, "core" is used. The core file need not be present.

Commands to *cdb* consist of an address, followed by a single command character, possibly followed by a command modifier. Usually if no address is given the last-printed address is used. An address may be followed by a comma and a number, in which case the command applies to the appropriate number of successive addresses.

Addresses are expressions composed of names, decimal numbers, and octal numbers (which begin with "0"), separated by "+" and "–". Evaluation proceeds left-to-right.

Names of external variables are written just as they are in C. For various reasons the external names generated by C all begin with an underscore, which is automatically tacked on by *cdb*. Currently it is not possible to suppress this feature, so symbols (defined in assembly-language programs) which do not begin with underscore are inaccessible.

Variables local to a function (automatic, static, and arguments) are accessible by writing the name of the function, a colon ":", and the name of the local variable (e.g. "main:argc"). There is no notion of the "current" function; its name must always be written explicitly.

A number which begins with "0" is taken to be octal; otherwise numbers are decimal, just as in C. There is no provision for input of floating numbers.

The construction "name[expression]" assumes that *name* is a pointer to an integer and is equivalent to the contents of the named cell plus twice the expression. Notice that *name* has to be a genuine pointer and that arrays are not accessible in this way. This is a consequence of the fact that types of variables are not currently saved in the symbol table.

The command characters are:

/*m* print the addressed words. *m* indicates the mode of printout; specifying a mode sets the mode until it is explicitly changed again:
- **o** octal (default)
- **i** decimal
- **f** single-precision floating-point
- **d** double-precision floating-point

\ Print the specified bytes in octal.

= print the value of the addressed expression in octal.

' print the addressed bytes as characters. Control and non-ASCII characters are escaped in octal.

" take the contents of the address as a pointer to a sequence of characters, and print the characters up to a null byte. Control and non-ASCII characters are escaped as octal.

& Try to interpret the contents of the address as a pointer, and print symbolically where the pointer points. The printout contains the name of an external symbol and, if required, the smallest possible positive offset. Only external symbols are considered.

? Interpret the addressed location as a PDP-11 instruction.

$*m* If no *m* is given, print a stack trace of the terminated or stopped program. The last call made is listed first; the actual arguments to each routine are given in octal. (If this is inappropriate, the arguments may be examined by name in the desired format using "/".) If *m* is "**r**", the contents of the PDP-11 general registers are listed. If *m* is "**f**", the contents of the floating-point registers are listed. In all cases, the reason why the program stopped or terminated is indicated.

%*m* According to *m*, set or delete a breakpoint, or run or continue the program:

b An address within the program must be given; a breakpoint is set there. Ordinarily, breakpoints will be set on the entry points of functions, but any location is possible as long as it is the first word of an instruction. (Labels don't appear in the symbol table yet.) Stopping at the actual first instruction of a function is undesirable because to make symbolic printouts work, the function's save sequence has to be completed; therefore *cdb* automatically moves breakpoints at the start of functions down to the first real code.

It is impossible to set breakpoints on pure-procedure programs (**−n** flag on *cc* or *ld* (I)) because the program text is write-protected.

d An address must be given; the breakpoint at that address is removed.

r Run the program being debugged. Following the "%r", arguments may be given; they cannot specify I/O redirection (">", "<") or filters. No address is permissible, and the program is restarted from scratch, not continued. Breakpoints should have been set if any were desired. The program will stop if any signal is generated, such as illegal instruction (including simulated floating point), bus error, or interrupt (see *signal*(II)); it will also stop when a breakpoint occurs and in any case announce the reason. Then a stack trace can be printed, named locations examined, etc.

c Continue after a breakpoint. It is possible but probably useless to continue after an error since there is no way to repair the cause of the error.

SEE ALSO

cc(I), db(I), *C Reference Manual* by D. M. Ritchie.

BUGS

Use caution in believing values of register variables at the lowest levels of the call stack; the value of a register is found by looking at the place where it was supposed to have been saved by the callee.

Some things are still needed to make *cdb* uniformly better than *db:* non-C symbols, patching files, patching core images of programs being run. It would be desirable to have the types of variables around to make the correct style printout more automatic. Structure members should be available.

Naturally, there are all sorts of neat features not handled, like conditional breakpoints, optional stopping on certain signals (like illegal instructions, to allow breakpointing of simulated floating-point programs).

NAME

chdir – change working directory

SYNOPSIS

chdir directory
cd directory

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *chdir* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

Cd is a synonym for *chdir* and acts identically.

SEE ALSO

sh(I), pwd(I)

NAME

chghist – change the history entry of an SCCS delta

SYNOPSIS

chghist –rSID name ...

DESCRIPTION

Chghist changes the history information, for the delta specified by the SID, of each named SCCS file.

If a directory is named, *chghist* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files, are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files, are silently ignored.

The exact permissions necessary to change the history entry of a delta are documented in the *SCCS/PWB User's Manual.* Simply stated, they are either (1) if you made a delta, you can change its history entry; or (2) if you own the file and directory you can change a history entry.

The new history is read from the standard input. If the standard input is a terminal (as determined by a successful *gtty*(II) call), the program will prompt (on the standard output) with "MRs? ", if the file has a **v** flag (see *admin*(I)), and with "comments? ". If the standard input is not a terminal, no prompt(s) is (are) printed. A newline preceded by a "\" is read as a blank, and may be used to make the entering of the history more convenient. The first newline not preceded by a "\" terminates the response for the corresponding prompt.

When the history entry of a delta table record (see *prt*(I)) is changed, all old MR entries (if any) are converted to comments, and both these and the original comments are preceded by a comment line that indicates who made the change and when it was made. The new information is entered preceding the old. No other changes are made to the delta table entry.

FILES

x-file	(see *delta*(I))
z-file	(see *delta*(I))

SEE ALSO

admin(I), get(I), delta(I), prt(I), help(I), sccsfile(V)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

chgrp – change group

SYNOPSIS

chgrp group file ...

DESCRIPTION

The group-ID of the files is changed to *group*. The group may be either a decimal GID or a group name found in the group-ID file.

SEE ALSO

chown(1), group(V)

FILES

/etc/group

NAME

chmod – change mode

SYNOPSIS

chmod octal file ...

DESCRIPTION

The octal mode replaces the mode of each of the files. The mode is constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit for shared, pure-procedure programs (see below)
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

Only the owner of a file (or the super-user) may change its mode.

If an executable file is set up for sharing ("**-n**" of *ld*(I)), then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worth while for heavily used commands.

SEE ALSO

ls(I), chmod(II)

NAME

chown – change owner

SYNOPSIS

chown owner file ...

DESCRIPTION

The user-ID of the files is changed to *owner*. The owner may be either a decimal UID or a login name found in the password file.

FILES

/etc/passwd

SEE ALSO

chgrp(I), passwd(V)

NAME

cmp – compare two files

SYNOPSIS

cmp [**–l**] [**–s**] 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. Moreover, return code 0 is yielded for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

Options:

–l Print the byte number (decimal) and the differing bytes (octal) for each difference.

–s Print nothing for differing files; return codes only.

SEE ALSO

diff(1), comm(1)

NAME

col – filter reverse line feeds

SYNOPSIS

col

DESCRIPTION

Col reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ASCII code ESC-7). *Col* is particularly useful for filtering multicolumn output made with the '.rt' command of *nroff*.

SEE ALSO

nroff(I)

BUGS

Can't back up more than 102 lines.

The input file must not have ASCII tab characters; *col* does not handle them properly (see *reform*(I)).

NAME

comb – combine SCCS deltas

SYNOPSIS

comb [**–o**] [**–s**] [**–psid**] [**–clist**] name ...

DESCRIPTION

Comb generates a shell procedure (see *sh*(I)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

–p The SCCS identification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

–c A list (see *get*(I) for the syntax of a list) of deltas to be preserved. All other deltas are discarded.

–o This argument causes the reconstructed file to be accessed at the release of the delta to be created for each "get –e" generated. Without this argument, the reconstructed file is accessed at the most recent ancestor for each "get –e" generated. Use of the **o** keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

–s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file, the file name, size after combining, original size, and pecentage change computed by:

100 * (original – combined) / original

(Sizes are in blocks.) We recommend that before any SCCS files are actually combined one should use this option to determine exactly how much space is saved by the combining process.

If no keyletter arguments are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

FILES

s.COMB	The name of the reconstructed SCCS file.
comb?????	Temporary.

SEE ALSO

get(I), delta(I), admin(I), prt(I), help(I), sccsfile(V), *SCCS/PWB User's Manual* by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

BUGS

Comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to actually be larger than the original.

NAME

comm – print lines common to two files

SYNOPSIS

comm [– [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be sorted in the same order, and produces a three column output: lines only in *file1;* lines only in *file2;* and lines in both files. The filename `–` means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm –12** prints only the lines common to the two files; **comm –23** prints only lines in the first file but not in the second; **comm –123** is a no-op.

SEE ALSO

cmp(I), diff(I), uniq(I)

NAME

cp — copy

SYNOPSIS

cp file1 file2

DESCRIPTION

The first file is copied onto the second. The mode and owner of the target file are preserved if it already existed; the mode of the source file is used otherwise.

If *file2* is a directory, then the target file is a file in that directory with the file-name of *file1*.

It is forbidden to copy a file onto itself.

SEE ALSO

cpx(I), ln(I), cat(I), pr(I), mv(I)

NAME

cpio – copy file archives in and out

SYNOPSIS

cpio –o[v]

cpio –i[drtuv] [pattern]

cpio –p[dlruv] [pattern] directory

DESCRIPTION

Cpio –o (copy out) reads the standard input for a list of pathnames and copies those files onto the standard output together with pathname and status information.

Cpio –i (copy in) extracts from the standard input, which is the product of a previous "*cpio –o*", files whose names are selected by a *pattern* given in the name-generating syntax of *sh*(I). The *pattern* meta-characters '?', '*', '[...]' will match '/' characters. The *pattern* argument defaults to "*".

Cpio –p (pass) copies out and in in a single operation. Destination pathnames are interpreted relative to the named *directory*.

The options are:

- **d** *Directories* are to be created as needed.
- **r** Interactively *rename* files. If the user types a null line, the file is skipped.
- **t** Print a *table of contents* of the input. No files are created.
- **u** Copy *unconditionally* (normally, an older file will not replace a newer file with the same name).
- **v** *Verbose:* causes a list of file names to be printed. When used with the t option, the table of contents looks like an "ls –l" (see *ls*(I)).
- **l** Whenever possible, link files rather than copying them. Usable only with the **–p** option.
- **m** Retain previous file modified time (only for the super-user).

The first example below copies the contents of a directory into an archive; the second duplicates a directory hierarchy:

```
ls | cpio –o >/dev/mt0

chdir olddir
find . –print | cpio –pdl newdir
```

SEE ALSO

ar(I), cpio(V)

BUGS

Path names are restricted to 128 characters.

If there are too many unique linked files, the program runs out of memory to keep track of them and subsequent linking information is lost.

NAME

cpx – copy a file exactly

SYNOPSIS

cpx – [file1 | –] [file2 | –]

DESCRIPTION

Cpx copies *file 1* onto *file 2*. The mode, owner and time of last modification of the source file are preserved.

Either *file1* or *file2* may be represented as a "–", which uses the standard UNIX input/output pipe mechanism, instead of the corresponding file. A file read from a pipe or written to a pipe will be preceded with a header, containing the mode, owner, time of last modification, number of characters, and a summed total of the characters in the file. The case where a pipe is read and a file is written, both the number of characters and the summed total are compared to similar values after the copy. If there are no differences between the comparisons, the message "ok" is printed.

Cpx prohibits copying a file onto itself.

Cpx does not allow *file1* to be a directory. If *file2* is a directory, then the target file is a file in that directory with the file name of *file1*.

Examples to copy a file to the current directory:

```
cpx ../file1 – | cpx – .
cpx ../file1 .
cpx ../file1 file2
```

SEE ALSO

cp(I)

NAME

cref – make cross reference listing

SYNOPSIS

cref [**−acilnostux123**] name ...

DESCRIPTION

Cref makes a cross reference listing of program files in assembler or C format. The files named as arguments in the command line are searched for symbols in the appropriate syntax.

The output report is in four columns:

(1)	(2)	(3)	(4)
symbol	file	see below	text as it appears in file

Cref uses either an *ignore* file or an *only* file. If the **−i** option is given, the next argument is taken to be an *ignore* file; if the **−o** option is given, the next argument is taken to be an *only* file. *Ignore* and *only* files are lists of symbols separated by new lines. All symbols in an *ignore* file are ignored in columns (1) and (3) of the output. If an *only* file is given, only symbols in that file appear in column (1). At most one of **−i** and **−o** may be used. The default setting is **−i**. Assembler predefined symbols or C keywords are ignored.

The **−s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, the current function name. The **−l** option causes the line number within the file to be put in column 3.

The **−t** option causes the next available argument to be used as the name of the intermediate temporary file (instead of /tmp/crt??). The file is created and is not removed at the end of the process.

Options:

- **a** assembler format (default)
- **c** C format input
- **i** use *ignore* file (see above)
- **l** put line number in col. 3 (instead of current symbol)
- **n** omit column 4 ("no context")
- **o** use *only* file (see above)
- **s** current symbol in col. 3 (default)
- **t** user supplied temporary file
- **u** print only symbols that occur exactly once
- **x** print only C external symbols
- **1** sort output on column 1 (default)
- **2** sort output on column 2
- **3** sort output on column 3

FILES

/tmp/crt??	temporaries
/usr/lib/aign	default assembler *ignore* file
/usr/lib/atab	grammar table for assembler files
/usr/lib/cign	default C *ignore* file
/usr/bin/crpost	post processor
/usr/lib/ctab	grammar table for C files

/usr/bin/upost post processor for **-u** option
/bin/sort used to sort temporaries

SEE ALSO
as(1), cc(1)

NAME

crypt – encode/decode

SYNOPSIS

crypt [password]

DESCRIPTION

crypt simulates a cryptographic machine.

crypt reads from the standard input file and writes on the standard output. It is thus suitable for use as a filter. For a given password, the encryption process is idempotent; that is,

crypt znorkle <clear >cypher
crypt znorkle <cypher

will print the clear.

NAME

csplit – context split

SYNOPSIS

csplit [**–s**] [**–f** prefix] file [RE01 RE02 ... REn]

DESCRIPTION

Csplit reads *file* and separates it into n+1 sections, defined by the regular expressions RE01, ... , REn, where n is less than 100. If the **–f** option is used, the sections are placed in *prefix*00 ... *prefix*n. The default is xx00 ... xxn. These sections get the following pieces of *file:*

00: from the start of the file up to (but not including) the first line matched by RE01
01: from the line matched by RE01 up to the first line that is matched by RE02
:
:
n+1: line matched by REn to the end of the file

Enclose by double quotes (") all RE's that contain blanks or other characters meaningful to the Shell.

Csplit tells the size of the original file, as well as of each "split" file as it creates it. It also prints any appropriate diagnostics. If the **–s** option is present, *csplit* suppresses the printing of all character counts.

EXAMPLE:

```
csplit –f zz file "procedure division" par5. par16.
```

After editing the "split" files, they can be recombined as follows:

```
cat zz0[0-3] >file
```

It should be noted that *csplit* does not affect in any way the original file. The responsibility for removing it is the user's.

SEE ALSO

ed(I), sh(I)

NAME

date – print and set the date

SYNOPSIS

date [mmddhhmm[yy]] [+format]

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. 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; *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. *Date* takes care of the conversion to and from local standard and daylight time.

If the argument begins with "+," the output of *date* is under the control of the user. The format for the output is similar to that of the first argument to *printf(III)*. All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by "%" and will be 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.

Field Descriptors:

n insert a newline character
t insert a tab character
m month of year – 01 to 12
d day of month – 01 to 31
y last 2 digits of year – 00 to 99
H hour – 00 to 23
M minute – 00 to 59
S second – 00 to 59
j julian date – 001 to 366
w day of week – Sunday = 0
a abbreviated weekday – Sun to Sat
h abbreviated month – Jan to Dec
r time in AM / PM notation

For example:

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

would generate as output:

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

DIAGNOSTICS

"No permission" if you aren't the super-user and you try to change the date; "bad conversion" if the date set is syntactically incorrect; "invalid option" if the field descriptor is not recognizable.

FILES

/dev/kmem

NAME

db – debug

SYNOPSIS

db [core [namelist]] [–]

DESCRIPTION

Unlike many debugging packages (including the Digital Equipment Corporation's ODT, on which *db* is loosely based), *db* is not loaded as part of the core image which it is used to examine; instead it examines files. Typically, the file will be either a core image produced after a fault or the binary output of the assembler. *Core* is the file being debugged; if omitted **core** is assumed. *Namelist* is a file containing a symbol table. If it is omitted, the symbol table is obtained from the file being debugged, or if not there from **a.out**. If no appropriate name list file can be found, *db* can still be used but some of its symbolic facilities become unavailable.

For the meaning of the optional third argument, see the last paragraph below.

The format for most *db* requests is an address followed by a one character command. Addresses are expressions built up as follows:

1. A name has the value assigned to it when the input file was assembled. It may be relocatable or not depending on the use of the name during the assembly.
2. An octal number is an absolute quantity with the appropriate value.
3. A decimal number immediately followed by '.' is an absolute quantity with the appropriate value.
4. An octal number immediately followed by **r** is a relocatable quantity with the appropriate value.
5. The symbol . indicates the current pointer of *db*. The current pointer is set by many *db* requests.
6. A * before an expression forms an expression whose value is the number in the word addressed by the first expression. A * alone is equivalent to '*.'.
7. Expressions separated by + or blank are expressions with value equal to the sum of the components. At most one of the components may be relocatable.
8. Expressions separated by – form an expression with value equal to the difference to the components. If the right component is relocatable, the left component must be relocatable.
9. Expressions are evaluated left to right.

Names for registers are built in:

r0 ... r5 sp pc fr0 ... fr5

These may be examined. Their values are deduced from the contents of the stack in a core image file. They are meaningless in a file that is not a core image.

If no address is given for a command, the current address (also specified by ".") is assumed. In general, "." points to the last word or byte printed by *db*.

There are *db* commands for examining locations interpreted as numbers, machine instructions, ASCII characters, and addresses. For numbers and characters, either bytes or words may be examined. The following commands are used to examine the specified file.

/ The addressed word is printed in octal.

\ The addressed byte is printed in octal.

" The addressed word is printed as two ASCII characters.

' The addressed byte is printed as an ASCII character.

` The addressed word is printed in decimal.

? The addressed word is interpreted as a machine instruction and a symbolic form of the instruction, including symbolic addresses, is printed. Often, the result will appear exactly as it was written in the source program.

& The addressed word is interpreted as a symbolic address and is printed as the name of the symbol whose value is closest to the addressed word, possibly followed by a signed offset.

<nl>(i. e., the character "new line") This command advances the current location counter "." and prints the resulting location in the mode last specified by one of the above requests.

^ This character decrements "." and prints the resulting location in the mode last selected one of the above requests. It is a converse to <nl>.

% Exit.

Odd addresses to word-oriented commands are rounded down. The incrementing and decrementing of "." done by the <nl> and ^ requests is by one or two depending on whether the last command was word or byte oriented.

The address portion of any of the above commands may be followed by a comma and then by an expression. In this case that number of sequential words or bytes specified by the expression is printed. "." is advanced so that it points at the last thing printed.

There are two commands to interpret the value of expressions.

= When preceded by an expression, the value of the expression is typed in octal. When not preceded by an expression, the value of "." is indicated. This command does not change the value of ".".

: An attempt is made to print the given expression as a symbolic address. If the expression is relocatable, that symbol is found whose value is nearest that of the expression, and the symbol is typed, followed by a sign and the appropriate offset. If the value of the expression is absolute, a symbol with exactly the indicated value is sought and printed if found; if no matching symbol is discovered, the octal value of the expression is given.

The following command may be used to patch the file being debugged.

! This command must be preceded by an expression. The value of the expression is stored at the location addressed by the current value of ".". The opcodes do not appear in the symbol table, so the user must assemble them by hand.

The following command is used after a fault has caused a core image file to be produced.

S causes the fault type and the contents of the general registers and several other registers to be printed both in octal and symbolic format. The values are as they were at the time of the fault.

For some purposes, it is important to know how addresses typed by the user correspond with locations in the file being debugged. The mapping algorithm employed by *db* is non-trivial for two reasons: First, in an **a.out** file, there is a 20(8) byte header which will not appear when the file is loaded into core for execution. Therefore, apparent location 0 should correspond with actual file offset 20. Second, addresses in core images do not correspond with the addresses used by the program because in a core

image there is a header containing the system's per-process data for the dumped process, and also because the stack is stored contiguously with the text and data part of the core image rather than at the highest possible locations. *Db* obeys the following rules:

If exactly one argument is given, and if it appears to be an **a.out** file, the 20-byte header is skipped during addressing, i.e., 20 is added to all addresses typed. As a consequence, the header can be examined beginning at location −20.

If exactly one argument is given and if the file does not appear to be an **a.out** file, no mapping is done.

If zero or two arguments are given, the mapping appropriate to a core image file is employed. This means that locations above the program break and below the stack effectively do not exist (and are not, in fact, recorded in the core file). Locations above the user's stack pointer are mapped, in looking at the core file, to the place where they are really stored. The per-process data kept by the system, which is stored in the first part of the core file, cannot currently be examined (except by **$**).

If one wants to examine a file which has an associated name list, but is not a core image file, the last argument "−" can be used (actually the only purpose of the last argument is to make the number of arguments not equal to two). This feature is used most frequently in examining the memory file /dev/mem.

SEE ALSO
as(I), core(V), a.out(V), od(I)

DIAGNOSTICS
"File not found" if the first argument cannot be read; otherwise "**?**".

BUGS
There should be some way to examine the registers and other per-process data in a core image; also there should be some way of specifying double-precision addresses. It does not know yet about shared text segments.

NAME

dc – desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number

The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.

+ – * % ^

The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the l is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged.

f All values on the stack and in registers are printed.

q exits the program. If executing a string, the recursion level is popped by two. If **q** is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x treats the top element of the stack as a character string and executes it as a string of dc commands.

[...] puts the bracketed ascii string onto the top of the stack.

<*x* >*x* =*x*

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation.

v replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! interprets the rest of the line as a UNIX 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.

o The top value on the stack is popped and used as the number radix for further output.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

? A line of input is taken from the input source (usually the console) and executed.

An example which prints the first ten values of n! is:

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

SEE ALSO

bc(I), which is a preprocessor for *dc* providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

(x) ? for unrecognized character x.
(x) ? for not enough elements on the stack to do what was asked by command x.
'Out of space' when the free list is exhausted (too many digits).
'Out of headers' for too many numbers being kept around.
'Out of pushdown' for too many items on the stack.
'Nesting Depth' for too many levels of nested execution.

NAME

dd – convert and copy a file

SYNOPSIS

dd [option=value] ...

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

option	*values*
if=	input file name; standard input is default
of=	output file name; standard output is default
ibs=	input block size (default 512)
obs=	output block size (default 512)
bs=	set both input and output block size, superseding *ibs* and *obs;* also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to *ibs*
... , ...	several conversions separated by commas

Where sizes are specified, a number of bytes is expected. A number may end with **k**, **b** or **w** to specify multiplication by 1024, 512, or 2 respectively; a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and new-line added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size *cbs.*

After completion, *dd* reports the number of whole and partial input and output blocks.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x:*

```
dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. *Dd* is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

SEE ALSO

cp(I)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. It is not clear how this relates to real life.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

NAME

delta – make an SCCS delta

SYNOPSIS

delta [**–s**] [**–n**] [**–rsid**] [**–glist**] [**–yhistory**] [**–mmrs**] [**–p**] name ...

DESCRIPTION

Delta adds a delta to each named SCCS file. If a directory is named, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files are silently ignored. (If a name of "–" is given the **y** keyletter must be present; see below.)

A *get* of many SCCS files, followed by a *delta* of those files should be avoided when the *get* generates a large amount of data. Instead, multiple *get*–*delta* sequences should be used.

Comments about the purpose of the delta(s) are supplied (once, and only once) either from the standard input, or by using the **y** argument. If one supplies the comments through the standard input, and the standard input is a terminal (as determined by a successful *gtty*(II) call), the program will prompt (on the standard output) with "comments? ". Otherwise, no prompt is printed. A newline preceded by a "\" may be used to make the entering of the comments more convenient. The first newline not preceded by a "\" terminates the comments response. The **y** argument is used to supply comments on the command line; if it is given the "comments?" question is not printed, and the standard input is not read.

If there is a *v* flag in the file (see *admin*(I)) the prompting is somewhat different. As the comments are solicited only once, if the first file processed has a *v* flag then all files processed must have a *v* flag (any files that don't will cause a diagnostic message and won't be processed; processing will continue with the next file). The inverse is also true.

When a file has a *v* flag, before prompting for "comments? " *delta* will prompt for "MRs? " (again, the prompt is only printed if the standard input is a terminal). MR numbers are read from the standard input separated by blanks and/or tabs. The same continuation rules apply as above. When an unadorned newline is read, *delta* will prompt for "comments? " as described above. If the *v* flag has a value, it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. This program is executed with the first argument having the value of the %M% identification keyword, a second argument of the value of the %Y% identification keyword, and third and subsequent arguments being the MR numbers. If a non-zero exit status is returned from this program *delta* will terminate (it is assumed that the MR numbers were not all valid). The **m** argument is used to supply MR numbers on the command line; if it is given the "MRs? " question is not printed, and the standard input is not read.

The following description is written as though only one SCCS file were named; the process of making a delta is equivalent for each file. (Note that the effects of any keyletter arguments apply independently to each SCCS file, and that the same comments are used for all files.)

The **g** argument specifies a list (see *get*(1) for the definition of <list>) of deltas which are to be marked *ignored* when the file is accessed at the change level created by this delta. (See the description of the *l-file* format in *get*(1)). A delta should only be ignored when the problem that caused the creation of the delta being ignored is no longer a problem at the change level created

by this delta.

The **p** argument causes *delta* to print the differences that constitute the delta on the standard output.

Delta makes a delta by "getting" the named file (see *get*(I)) at the SID specified by the **r** keyletter (this SID *must* be listed in the *p-file*), or at the same SID that was used when the *get* command was executed with the **e** argument by the user executing *delta* (if the user executing *delta* is listed more than once in the *p-file*, the **r** argument *must* be supplied). The "gotten" file is then compared with the *g-file*; the differences between the two files constitute the delta.

When the comparison is finished, *delta* prints the SID of the new delta, followed by the number of lines inserted, deleted, and unchanged. The **s** argument suppresses this printing. Normally, the *g-file* is removed after the delta is made. The **n** argument suppresses the removal.

Delta will ignore hangups if it is already ignoring interrupts.

FILES

g-file	See *get* for an explanation of the *g-file*.
p-file	Information from *get*.
x-file	Replacement for the SCCS file. The naming convention is the same as that for the *p-file* (see *get*).
z-file	Lockout file; see *get*(I).
d-file	"Gotten" file; temporary. The naming convention is the same as that for the *p-file* (see *get*).
/usr/bin/bdiff	Program to compute differences between the "gotten" file and the *g-file*.

SEE ALSO

get(I), admin(I), prt(I), help(I), sccsfile(V), bdiff(I)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

deroff – remove nroff, troff, and eqn constructs

SYNOPSIS

deroff [**–w**] file ...

DESCRIPTION

Deroff reads each file in sequence and removes all *nroff* and *troff* command lines, backslash constructions, macro definitions, and equations (between ".EQ" and ".EN" lines or between delimiters) and writes the remainder on the standard output. *Deroff* follows chains of included files (".so" and ".nx" commands); if a file has already been included, a ".so" is ignored and a ".nx" terminates execution. If no input file is given, *deroff* reads from the standard input file.

If the **–w** flag is given, the output is a word list, one "word" (string of letters, digits, and apostrophes, beginning with a letter; apostrophes are removed) per line, and all other characters ignored. Otherwise, the output follows the original, with the deletions mentioned above.

SEE ALSO

nroff(I), troff(I), eqn(I)

DIAGNOSTICS

Complains if a file cannot be opened.

BUGS

Does not handle recursive backslash constructions like \h'\w'c''.

NAME

df – report disk free space

SYNOPSIS

df [**–uqs**] [**–**tnumber] [arg ...]

DESCRIPTION

Df prints the number of free blocks on a file system. If no *args* are specified, the free counts of all the mounted file systems are printed.

The **–u** flag prints the total block size, number of blocks allocated for system information, number of free blocks, number of blocks used and the number of free inodes.

The **–q** flag determines and prints the number of free blocks on a file system by extracting the free count directly from the file system's superblock.

The **–s** flag is a silent option which prohibits printing of any results. Error messages and exit status are not effected.

The **–t** flag followed by a decimal *number* (5 digit maximum) is compared with the number of free blocks. The result of the comparison returns the file system's major and minor device numbers and a single character either **Y** or **N**, to indicate if the number of free blocks is greater or less than the requested *number*, respectively (e.g., df –t 1000 /u8 returns "0 12 Y"). An exit status of 0 is returned for **Y** and 1 for **N**.

The *arg* can be specified as either the root name of the **mounted** file system, e.g., "/u8" or the name of the special file corresponding to the particular device (must refer to a disk), e.g., "/dev/rp14".

FILES

/dev/rf?, /dev/rk?, /dev/rp?, /etc/mnttab

SEE ALSO

icheck(VIII)

NAME

diff – differential file comparator

SYNOPSIS

diff [**–efb**] name1 name2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If *name1 (name2)* is '–', the standard input is used. If *name1 (name2)* is a directory, then a file in that directory whose file-name is the same as the file-name of *name2 (name1)* 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 file *name1* into file *name2*. The numbers after the letters pertain to file *name2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert file *name2* into *name1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by '<', then all the lines that are affected in the second file flagged by '>'.

The **–b** option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal. The **–e** option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate file *name2* from file *name1*. 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.

```
(cat $2 ... $9; echo '1,$p') | ed – $1
```

Except for occasional 'jackpots', *diff* finds a smallest sufficient set of file differences.

FILES

/tmp/d?????

SEE ALSO

cmp(I), comm(I), ed(I), uniq(I)

DIAGNOSTICS

'jackpot' – To speed things up, the program uses hashing. You have stumbled on a case where there is a chance that this has resulted in a difference being called where none actually existed. Sometimes reversing the order of files will make a jackpot go away.

BUGS

Editing scripts produced under the **–e** or **–f** options are naive about creating lines consisting of a single '.'.

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**–ex3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*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 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 will incorporate 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
```

SEE ALSO

diff(I)

BUGS

Text lines that consist of a single '.' will defeat **–e**.

FILES

/tmp/d3a?????
/tmp/d3b?????

NAME

diffmark – mark changes between versions of a file

SYNOPSIS

diffmark [**code**"*string*"] ... [*name*]

DESCRIPTION

Diffmark is a filter used to modify the editor command version of *diff(I)* output so that it can be used to mark the changes between successive versions of a file. Its most common use is to automatically insert change mark commands into a file of text for *nroff(I)* or *troff(I)*. The following is a typical command sequence:

diff –e oldfile newfile | diffmark markedfile | ed – oldfile

The generated file *markedfile* is the same as *newfile*, except that it has the needed change mark requests inserted. The user would normally print *markedfile*, and later remove it and *oldfile* when no longer needed.

Diffmark adds extra lines to the output of *diff*. It inserts one line at both the beginning and end of each sequence of appended or changed lines, and appends two lines following each deletion. The default values of these lines are chosen to make use of the "margin character" request of the formatters. The user may override any such value by supplying an option string, which is concatenated with a newline to make up the line. Any null option string causes its corresponding line to be omitted completely. The option codes and their defaults are as follows:

–ab".mc |" – "append" beginning – insert at beginning of an addition

–ae".mc" – "append" end – insert at end of an addition

–cb".mc |" – "change" beginning – insert at beginning of a change

–ce".mc" – "change" end – insert at end of a change

–db".mc *" – "delete" 1st line – insert as first line of deletion

–de".mc" – "delete" 2nd line – insert as 2nd line of deletion

Although not a necessity, the following option is convenient:

name causes *diffmark* to append "w *name*" to the end of its output. For safety's sake, this name should *not* be the same as that of the file being edited.

Here is an example. Suppose you run the following sequence of commands:

```
diff –e oldfile newfile >diff1
diffmark diff3 –cb".mc +" <diff1 >diff2
ed – oldfile <diff2
nroff diff3 >diff4
```

Of course, the only reason for doing this rather than using pipelines is to see what all the files look like:

oldfile:
```
.nf
ccc
eee
ggg
hhh
zzz
```

newfile:
```
.nf
aaa
eee
fff
ggg
zzz
```

diff1 (output from diff):
```
5d
3a
fff
.
2c
aaa
.
```

diff2 (output from diffmark):
```
5c
.mc *
.mc
.
3a
.mc |
fff
.mc
.
2c
.mc +
aaa
.mc
.
w diff3
```

diff3 (edited version of oldfile):
```
.nf
.mc +
aaa
.mc
eee
.mc |
fff
.mc
ggg
```

```
.mc *
.mc
zzz
```

diff4 (formatted output, with line length set to 10):

```
aaa        +
eee
fff        |
ggg
zzz        *
```

If you are so inclined, you can use *diffmark* to produce listings of C (or other) programs with changes marked. A typical shell procedure is:

```
:          cdiffmk: shell proc to show C program differences
:          called: cdiffmk old new
diff -c $1 $2 | (diffmark;echo '1,$p') | ed - $1 | nroff macs - | pr -h $2
```

The file macs looks like this:

```
.pll
.ll 77
.nf
.eo
.nc
```

The **ll** request might specify a different line length, depending on the nature of the program being printed. The **eo** and **nc** requests are probably needed only for C programs.

DIAGNOSTICS

"input not from diff"
"line too long" (>512 characters)
Up to 72 characters of the offending line are printed immediately following the diagnostic.

EXIT CODES

0 – normal completion
1 – input did not appear to be from *diff*, or other error

SEE ALSO

diff(I), nroff(I), troff(I)

BUGS

Esthetic considerations may dictate manual adjustment of some output. File differences involving only formatting requests may produce undesirable output. I.e., replacing ".sp" by ".sp2" will produce a change mark on the preceding or following line of output.

For those who use *diffmark* to produce UNIX Manual pages, extra handling may be needed to get vertical bars to appear. This results from the choice of the bar as the character translated to a nonadjustable blank for use with tabs. When you use *diffmark*, override the default choice of "|" by "!" instead, causing the latter to appear in your final output. If you prefer the vertical bar, you can get it on the final output by adding the following to the beginning of your file:

```
.if n .tr !|
.if n .ds v !
```

which may be mysterious, but works.

NAME

dsw — delete interactively

SYNOPSIS

dsw [directory]

DESCRIPTION

For each file in the given directory ('.' if not specified) *dsw* types its name. If y is typed, the file is deleted; if **x**, *dsw* exits; if new-line, the file is not deleted; if anything else, *dsw* asks again.

SEE ALSO

rm(I)

BUGS

The name *dsw* is a carryover from the ancient past. Its etymology is amusing.

NAME

du – summarize disk usage

SYNOPSIS

du [**–s**] [**–a**] [name ...]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each specified directory or file *name*. If *name* is missing, '.' is used.

The optional argument **–s** causes only the grand total to be given. The optional argument **–a** causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

A file which has two links to it is only counted once.

BUGS

Non-directories given as arguments (not under **–a** option) are not listed.

Removable file systems do not work correctly since i-numbers may be repeated while the corresponding files are distinct. *Du* should maintain an i-number list per root directory encountered.

NAME

echo – echo arguments

SYNOPSIS

echo [arg ...]

DESCRIPTION

Echo writes its arguments in order as a line on the standard output file. It is mainly useful for producing diagnostics in command files.

Certain escape sequences are recognized:

"\n" causes the newline character to be written.

"\c" terminates *echo* without a newline.

"\0N" causes the octal number *N* to be written.

SEE ALSO

pump(I)

NAME

ed – text editor

SYNOPSIS

ed [–] [+] [name]

DESCRIPTION

Ed is the standard text editor.

If the *name* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed's* buffer so that it can be edited. The optional '–' suppresses the printing of character counts by *e*, *r*, and *w* (or *z*) commands.

Ed operates on a copy of the file it is editing; changes made in the copy have no effect on the file until a *w* or *z* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*. There is only one buffer.

If changes have been made in the buffer since the last *w* or *z* command, *ed* warns the user if an attempt is made to destroy *ed's* buffer via the *q* or *e* commands. *Ed* prints 'q?' or 'e?', respectively, and allows one to continue editing. A second *q* or *e* command at this point will take effect. This warning feature may be inhibited by specifying the '+' option (e.g., ed + file). The '–' option also inhibits this feature.

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 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. Input mode is left by typing a period '.' alone at the beginning of a line.

Ed 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 replaced. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. The regular expressions allowed by *ed* are constructed as follows:

The following *one-character regular expressions* match a single character:

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

1.2 A backslash '\' followed by any special character is a one-character regular expression that matches the special character itself. The special characters are:

a. '.', '*', '+', '[', and '\' (period, asterisk, plus sign, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets '[]' (see 1.4 below).

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

c. '$' (currency symbol), which special at the end of an entire regular expression (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire regular expression, which is special for that regular expression (for example, see how '/' is used in the *g* command, below.)

1.3 A period '.' is a one-character regular expression that matches any character except the new-line character.

1.4 A non-empty string of characters enclosed in square brackets '[]' is a one-character regular expression that matches *any one* character in that string. If, however, the first character of the string is a circumflex '^', the one-character regular expression matches any character *except* new-line 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 ']' does not terminate such a string when it occurs first (after an initial '^', if any), in it, e.g., '[]a]' matches either a right square bracket ']' or the letter 'a'. The five characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *regular expressions* from *one-character regular expressions*:

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

2.2 A one-character regular expression followed by an asterisk '*' is a regular expression that matches *zero* or more occurrences of the one-character regular expression. If there is any choice, this regular expression matches as many occurrences as possible.

2.3 A one-character regular expression followed by a plus '+' is a regular expression that matches *one* or more occurrences of the one-character regular expression. If there is any choice, this regular expression matches as many occurrences as possible.

2.4 A one-character regular expression followed by '\{ *m*\}', '\{ *m*,\}', or '\{ *m*,*n*\}' is a regular expression that matches a *range* of occurrences of the one-character regular expression. 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 regular expression matches as many occurrences as possible.

2.5 The concatenation of regular expressions is a regular expression that matches the concatenation of the strings matched by each component of the regular expression.

2.6 A regular expression enclosed between the character sequences '\(' and '\)' is a regular expression that matches whatever the unadorned regular expression matches; this construction has side effects discussed under the *s* command, below.

Finally, an *entire regular expression* 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 regular expression constrains that regular expression to match an *initial* segment of a line.

3.2 A currency symbol '$' at the end of an entire regular expression constrains that regular expression to match a *final* segment of a line. The construction *^entire regular expression$* constrains the entire regular expression to match the entire line.

The null regular expression standing alone (e.g., '//') is equivalent to the last regular expression encountered.

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 the command. Addresses are constructed as follows:

1. The character '.' addresses the current line.
2. The character '$' addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. ''*x*' addresses the line marked with the mark name character *x*, which must be a lower-case letter. Lines are marked with the *k* command described below.
5. A regular expression 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 regular expression. If necessary, the search wraps around to the beginning of the buffer and continues through the current line, so that the entire buffer is searched.
6. A regular expression enclosed in queries '?' 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 regular expression. If necessary the search wraps around to the end of the buffer and continues through the current line.
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 or subtracted, respectively. As a consequence of this rule and of rule 8, the address '−' refers to the line preceding the current line. Moreover, trailing '+' and '−' characters have a cumulative effect, so '−−' refers to the current line less 2.
10. To maintain compatibility with earlier versions of the editor, the character '^' in addresses is entirely equivalent to '−'.

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) are 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 before the second address is interpreted. This feature can be used to determine the starting line for forward and backward searches (see items 5. and 6. in the list above). 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, but are used to 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 may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed, respectively, as discussed below under the *p* and *l* 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: text is placed at the beginning of the buffer.

(.,.)c
<text>
.

The *c*hange command deletes the addressed lines, then accepts input text which replaces these lines. '.' is left at the last line input; if there were none, it is left at the first line 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 name

The *e*dit command causes the entire contents of the buffer to be deleted, and then the named file to be read in; '.' is set to the last line of the buffer. If no file name is given, the remembered file name, if any, is used (see the *f* command). The number of characters read is typed; *name* is remembered for possible use as a default file name in subsequent *e* or *r* or *w* or *z* commands.

f name

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

(1,$)g/regular expression/command list

In the *g*lobal command, the first step is to mark every line that matches the given *regular expression*. Then, for every such line, the given *command list* is executed with '.' initially set to that line. A single command or the first of 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 are permitted; the '.' terminating input mode may be omitted if it would be the last line of the *command list*. The (global) commands (*g*, *v*, *G*, and *V*) are *not* permitted in the *command list*.

(.)h

The date as returned by *date*(I) is appended after the addressed line.

(.)i
<text>
.

The *i*nsert command inserts the given text before the addressed line. '.' is left at the last inserted line; or, if there were none, at the addressed line. This command differs from the *a* command only in the placement of the input text.

(.,.+1)j

The *j*oin command joins contiguous lines by removing the appropriate new-line characters.

(.)k*x*

The mar*k* command marks the addressed line with name *x*, which must be a lower-case letter. The address form '*'x*' then addresses this line.

(.,.)l

The *l*ist command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., *tab, backspace*) are represented by (hopefully) mnemonic overstrikes, all other non-printing characters are printed in octal, and long lines are folded. An *l* command may also be appended to any other command.

(.,.)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. The last line so moved becomes the current line.

(.,.)p

The *p*rint command prints the addressed lines; '.' is left at the last line printed. The *p* command may be appended to any other command (e.g., '*dp*' deletes the current line and prints the new current line).

q

The *q*uit command causes *ed* to exit. No automatic write of a file is done.

($)r name

The *r*ead command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The remembered file name is not changed unless *name* is the very first file name 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.

(.,.)s/regular expression/replacement/ or,
(.,.)s/regular expression/replacement/g

The *s*ubstitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all (non-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. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of '/' to delimit the regular expression and the *replacement;* '.' is left at the last line on which a substitution occurred.

An ampersand '&' appearing in the *replacement* is replaced by the string matching the regular expression 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 regular expression enclosed between '\(' and '\)'. When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of '\(' starting from the left.

A line may be split by substituting a new-line character into it. The new-line in the *replacement* must be escaped by preceding it by '\'. 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'); '.' is left at the last line of the copy.

u

This command reverses the effect of the last *s* command. The *u* command affects only the last line changed by the most recent *s* command.

(1,$) v/regular expression/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 *regular expression.*

(1,$) w name
(1,$) z name
The *write* command writes the addressed lines onto the named file. If the file does not exist, it is created with mode 644 (readable by everyone, writable by you). The remembered file name is *not* changed unless *name* is the very first file name mentioned since *ed* was invoked. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands); '.' is unchanged. If the command is successful, the number of characters written is typed. The *z* command is identical to *w* but, on most keyboards, the 'z' key is farther from the 'q' key than is the 'w' key.

(1,$) G/regular expression/
In the interactive *G*lobal command, the first step is to mark every line that matches the given *regular expression.* Then, for every such line, that line is printed, '.' is changed to that line, and any *one* command, other than a global command (*g, v, G,* and *V*), must be input. 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 this invocation of *G.* Note that the commands input after the *G* command prints each marked line may address and affect *any* lines in the buffer. The *G* command can be terminated by an interrupt signal (ASCII DEL or BREAK).

P
The editor will prompt with a '*' for all subsequent commands. This command alternately turns the mode on and off; it is initially off.

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

(1,$) V/regular expression/
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 *regular expression.*

($) =
The line number of the addressed line is typed; '.' is unchanged by this command.

!UNIX command
The remainder of the line after the '!' is sent to the UNIX shell (*sh*(I)) to be interpreted as a command; '.' is unchanged.

(.+1) <new-line>
An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1p'; it is useful for stepping through text.

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

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name, and 128K characters in the buffer. The limit on the number of lines depends on the amount of user memory: each line takes 1 word.

Ed allows the user to include, in the first line of each text file, a specification to control the line length and the tab-to-space conversion. For example, <:t5,10,15s72:> sets tabs at columns 5, 10, and 15; it will also truncate the *printing* of all lines to a length of 72 characters and warn when a truncation has occurred. For the specification to take effect, the user's terminal must be in *echo*

and *−tabs* modes (see *stty*(I)). Only the 't' and 's' parameters may be used as described in *fspec*(V). If the 's' parameter is used, all referenced lines are checked for maximum length on file read and write operations and on line print operations. Appropriate diagnostics are generated. Truncation occurs *only* on printing.

If the user attempts a *w* or *z* command and the destination file system does not have enough space available, a diagnostic is printed with an error number (i.e. "NO SPACE: e1"). *Ed* will not perform the write. The *UNIX* command "help e1" (see *help*(I)) prints out a full description of what to do. *Help* should be executed before leaving the editor (e.g., "!help e1").

FILES

/tmp/e#, temporary; '#' is the process number (in octal).

DIAGNOSTICS

'?' for errors in commands; 'TMP?' for temporary file (buffer) overflow; *help*(I) error numbers in all other cases. Commands in error should be re-entered properly. On temporary file overflow, the buffer should be written to a file and then an *e* command executed on that file. This will re-initialize the buffer; note that if the buffer overflows during the execution of a command that, in the absence of the TMP? diagnostic, would have done several changes, only some of the changes may have been done. Help error messages are self-explanatory.

SEE ALSO

A Tutorial Introduction to the UNIX Text Editor by B. W. Kernighan.
Advanced Editing on UNIX by B. W. Kernighan.

BUGS

If the *s* command succeeds on (i.e., modifies) a line that was marked by a *g, v, G,* or *V* command, then that mark is effectively removed. The editor deletes all ASCII *null* characters whenever it reads text into the buffer.

NAME

egrep – search a file for lines containing a pattern

SYNOPSIS

egrep [**–b**] [**–c**] [**–f**] [**–n**] [**–v**] pattern [file] ...

DESCRIPTION

egrep searches the input files (standard input default) for all lines containing an instance of the regular expression *pattern.* Normally, each line matched is copied to the standard output. The *pattern* matches a line whenever the line contains a substring denoted by the *pattern.*

The flags modify the normal behavior as follows:

–b causes each printed line to be preceded by the block number on which it was found
–c causes only a count of matching lines to be printed
–f causes the regular expression to come from a file named *pattern*
–n causes each printed line to be preceded by its relative line number in the file
–v causes all lines but those matching the *pattern* to be printed

In all cases the file name is shown if there is more than one input file.

A *pattern* is one of the following:

1. an ordinary character (denoting itself)
2. a circumflex '^' (denoting the beginning of a line)
3. a dollar sign '$' (denoting the end of a line)
4. a period '.' (denoting any character but a newline)
5. '[' followed by a string of characters followed by ']' (denoting any character in the string; if the first character in this string is '^', the pattern denotes any character except newline and the characters in the string)
6. '(' followed by a pattern followed by ')' (denoting the enclosed pattern)
7. a pattern followed by '*', or by '+', or by '?' (denoting zero or more, one or more, or zero or one instances, respectively, of the preceding pattern)
8. a pattern followed by a pattern (denoting concatenation of the two patterns)
9. a pattern followed by '|' followed by another pattern (denoting the alternation of the two patterns); a newline may be used in place of '|'.

In parsing a pattern, the rules are applied in the order given.

A pattern metacharacter can be used as an ordinary character by preceding it by '\'. The metacharacters are: '^', '$', '.', '[', ']', '*', '+', '?', '(', ')', '\'.

Care should be taken when using the characters $ * [^ | ( ) and \ in the regular expression as they are also meaningful to the Shell. When *pattern* is a regular expression other than a simple string, it is generally necessary to enclose the entire *pattern* argument in quotes.

SEE ALSO

grep(I), fgrep(I), lex(I), rgrep(I), sed(I), ed(I), sh(I)

BUGS

Lines longer than 512 characters are not printed completely.

NAME

eqn – typeset mathematics

SYNOPSIS

eqn [file] ...

DESCRIPTION

Eqn is a *troff*(I) preprocessor for typesetting mathematics on the Graphics Systems, Inc. phototypesetter. Usage is almost always

eqn file ... | troff

If no files are specified, *eqn* reads from the standard input. A line beginning with ".EQ" marks the start of an equation; the end of an equation is marked by a line beginning with ".EN". Neither of these lines is altered or defined by *eqn*, so you can define them yourself in *troff*(I) to get centering, numbering, etc. All other lines are treated as comments, and passed through untouched.

Spaces, tabs, new-lines, braces, double quotes, tilde, and circumflex are the only delimiters. Braces "{}" are used for grouping. Use tildes "~" to get extra spaces in an equation.

Subscripts and superscripts are produced with the keywords **sub** and **sup.** Thus *x sub i* makes x_i, *a sub i sup 2* produces a_i^2, and *e sup {x sup 2 + y sup 2}* gives $e^{x^2+y^2}$. Fractions are made with **over.** *a over b* is $\frac{a}{b}$ and *1 over sqrt {ax sup 2 +bx+c}* is $\frac{1}{\sqrt{ax^2+bx+c}}$; **sqrt** makes square roots.

The keywords **from** and **to** introduce lower and upper limits on arbitrary things: $\lim_{n\to\infty} \sum_0^n x_i$ is made with *lim from {n-> inf} sum from 0 to n x sub i.* Left and right brackets, braces, etc., of the right height are made with **left** and **right:** *left [x sup 2 + y sup 2 over alpha right] ~=~1* produces $\left[x^2+\frac{y^2}{\alpha}\right] = 1$. The **right** clause is optional.

Vertical piles of things are made with **pile, lpile, cpile,** and **rpile:** *pile {a above b above c}* produces

a
b. There can be an arbitrary number of elements in a pile. **lpile** left-justifies, **pile** and **cpile**
c

center, with different vertical spacing, and **rpile** right justifies.

Diacritical marks are made with **dot, dotdot, hat, bar:** *x hat = f(t) bar* is $\hat{x}=\overline{f(t)}$. Default sizes and fonts can be changed with **size n** and various of **roman, italic,** and **bold.**

Keywords like *sum* ($\sum$), *int* ($\int$), *inf* (∞), and shorthands like >=, ($\geqslant$), ->, ($\rightarrow$), !=, ($\neq$) are recognized. Spell out Greek letters in the desired case, as in *alpha, GAMMA.* Mathematical words like *sin, cos, log* are made Roman automatically. *Troff*(I) four-character escapes like \(ua (↑ – for "up arrow") can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched.

SEE ALSO

Typesetting Mathematics – User's Guide (2nd Edition) by B. W. Kernighan and L. L. Cherry
New Graphic Symbols for EQN and NEQN by C. Scrocca
NROFF/TROFF User's Manual by J. F. Ossanna
troff(I), neqn(I)

BUGS

Undoubtedly. Watch out for small or large point sizes – it's tuned too well for size 10. Be cautious if inserting horizontal or vertical motions, and of backslashes in general.

NAME

= (equals) – shell assignment command

SYNOPSIS

= letter [arg1 [arg2]]

DESCRIPTION

The "=" command provides shell string variables. The 26 *letter* variables ('a'–'z') are referenced in later commands in the manner of shell arguments, i.e.: *$a*, ... *$z*. If no arguments are given, the standard input is read to newline or EOT for the value. The exit code is set to 0 if a newline is found in the input; it is set to 1 otherwise, thus providing and end-of-file indicator. If *arg1* is the only argument, or if two non-null arguments are given, the variable is set to *arg1*, and the exit code set to 0. If two arguments are given, and if *arg1* is a null string, the value of *arg2* is assigned to the variable, and the exit code is set to 1, permitting a convenient default mechanism:

```
= a "$1" "default value" && shift
```

The "=" command works either at the terminal, or in shell command files. The variables can be assigned repeatedly. Storage is assigned as needed, but there is no recovery.

Some *letter* variables have predefined meanings and are initialized once at the time the Shell begins execution:

$n The argument count. "sh file arg1 arg2 arg3" has the value 3. The shift command does not change the value of $n.

$p This variable holds the shell search sequence of pathname prefixes for command execution. Alternatives are separated by ":". The default initial value is:

```
= p ":/bin:/usr/bin"
```

which prepends successively
the null pathname (execute from current dir.),
/bin,
/usr/bin.

Using the same type of specification, users may choose their own sequence by storing it in a file named ".path" in their login directory. The ".path" information passes to successive shells (and other commands like *time*(I) or *nohup*(I)); the $p value does not. In any case, no prepending occurs when a command name contains a '/'.

$r exit(status) of the most recent command executed by the Shell. The value is ASCII numeric, and is initially '0'. At end-of-file the shell exits with the value of *$r*.

$s Name of login (starting) directory.

$t Terminal identification letter or number: /dev/tty$t is a file name for the terminal.

$w First component in $s pathname, i.e., file system name (such as /usr).

$z Is the name of the Shell. Its default value is '/bin/sh', but this can be overridden by supplying a name as the second line of the '.path' file.

Note that variables ('a' – 'm') are guaranteed to be initialized to null strings and usable in any way desired. Variables ('n' – 'z') may acquire special uses in the future. The values of $n, $s, $t, and $w may reasonably be modified; it is catastrophic to change $p; it is possible, but useless to modify $r.

The "=" command is executed within the shell. Note that it is commonly used to read the first line of output from a pipe or a line from the terminal, for example:

```
grep -c string file | = a
```

or:

```
= a </dev/tty
```

EXIT CODES

0 – normal read, or first of two arguments is not null.
1 – end-of-file, or first of two arguments is null.

SEE ALSO

expr(I), sh(I)

NAME

exit – terminate command file

SYNOPSIS

exit [integer]

DESCRIPTION

Exit performs a **seek** to the end of its standard input file. Thus, if it is invoked inside a file of commands, upon return from *exit* the shell will discover an end-of-file and terminate.

The optional argument will be returned to the shell as the exit status code.

SEE ALSO

if(I), goto(I), sh(I)

NAME

expr – evaluate arguments as an algebraic expression

SYNOPSIS

expr arg ...

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, i.e., '*', '|', '&', '(', and ')', must be escaped.

The operators and keywords are listed below. 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*

expr \& *expr*

expr { +, – } *expr*

expr { *, /, % } *expr*

substr *expr expr expr*

length *expr*

index *expr expr*

\(*expr* \)

The result of *substr* is that portion of the first expression (possibly null) which is defined by the offset (second expression, starting at '1') and the span (third expression). A large span value can be given to obtain the remainder of the string.

Length returns the length in characters of the expression that follows.

Index searches the first expression for the first character that matches a character from the second expression. It returns the character position number if it succeeds, or '0' if it fails.

The *expr* command is handy with Shell variables. For example:

```
expr substr xxx$a "(" length xxx$a – 2 ")" 3 | = b
```

assigns the last three characters of the Shell variable *$a* into the variable *$b*.

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.

DIAGNOSTICS

Grumbles from *yacc*(I) for syntax violations.
"Non-numeric argument" if the argument needs to be, but is not, an integer.

NAME

fc – Fortran compiler

SYNOPSIS

fc [**–c**] sfile1.f ... ofile1 ...

DESCRIPTION

Fc is the UNIX Fortran compiler. It accepts three types of arguments:

Arguments whose names end with '.f' are assumed to be Fortran source program files; they are compiled, and the object program is left on the file 'sfile1.o' (i.e., the file whose name is that of the source with '.o' substituted for '.f').

Other arguments (except for **–c**) are assumed to be either loader flags, or object programs, typically produced by an earlier *fc* run, or perhaps libraries of Fortran-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

The **–c** argument suppresses the loading phase, as does any syntax error in any of the routines being compiled.

The following is a list of differences between *fc* and ANSI standard Fortran (also see the BUGS section):

1. Arbitrary combination of types is allowed in expressions. Not all combinations are expected to be supported at runtime. All of the normal conversions involving integer, real, double precision and complex are allowed.
2. Two forms of "implicit" statements are recognized: **implicit integer /i–n/** or **implicit integer (i–n).**
3. The types **doublecomplex, logical*1, integer*1, integer*2, integer*4** (same as **integer**), **real*4 (real)**, and **real*8 (double precision**) are supported.
4. **&** as the first character of a line signals a continuation card.
5. **c** as the first character of a line signals a comment.
6. All keywords are recognized in lower case.
7. The notion of 'column 7' is not implemented.
8. G-format input is free form; leading blanks are ignored, the first blank after the start of the number terminates the field.
9. A comma in any numeric or logical input field terminates the field.
10. There is no carriage control on output.
11. A sequence of *n* characters in double quotes '"' is equivalent to *n* **h** followed by those characters.
12. In **data** statements, a hollerith string may initialize an array or a sequence of array elements.
13. The number of storage units requested by a binary **read** must be identical to the number contained in the record being read.
14. If the first character in an input file is "#", a preprocessor which implements "#define" and "#include" preprocessor statements is called. These preprocessor statements are similar to the corresponding C preprocessor statements; see the C reference manual for details. The

preprocessor does not recognize Hollerith strings written with *n*h.

In I/O statements, only unit numbers 0-19 are supported. Unit number *n* refers to file fort*nn;* (e.g. unit 9 is file 'fort09'). For input, the file must exist; for output, it will be created. Unit 5 is permanently associated with the standard input file; unit 6 with the standard output file. Also see *setfil*(III) for a way to associate unit numbers with named files.

FILES

a.out	loaded output
f.tmp[123]	temporary (deleted)
/usr/fort/errors	compile-time error messages
/usr/fort/fc1	compiler proper
/lib/fr0.o	runtime startoff
/lib/filib.a	interpreter library
/lib/libf.a	builtin functions, etc.
/lib/liba.a	system library

SEE ALSO

rc(I), which announces a pleasant Fortran dialect; the ANSI standard; *ld*(I) for loader flags. For some subroutines, try *ierror, getarg, setfil*(III).

DIAGNOSTICS

Compile-time diagnostics are given in English, accompanied if possible with the offending line number and source line with an underscore where the error occurred. Runtime diagnostics are given by number as follows:

1 invalid log argument
2 bad arg count to amod
3 bad arg count to atan2
4 excessive argument to cabs
5 exp too large in cexp
6 bad arg count to cmplx
7 bad arg count to dim
8 excessive argument to exp
9 bad arg count to idim
10 bad arg count to isign
11 bad arg count to mod
12 bad arg count to sign
13 illegal argument to sqrt
14 assigned/computed goto out of range
15 subscript out of range
16 real**real overflow
17 (negative real)**real
100 illegal I/O unit number
101 inconsistent use of I/O unit
102 cannot create output file
103 cannot open input file
104 EOF on input file
105 illegal character in format
106 format does not begin with (
107 no conversion in format but non-empty list
108 excessive parenthesis depth in format
109 illegal format specification

110 illegal character in input field
111 end of format in hollerith specification
112 bad argument to setfil
120 bad argument to ierror
999 unimplemented input conversion

BUGS

The following is a list of those features not yet implemented:
arithmetic statement functions
scale factors on input
Backspace statement.

NAME

fd2 – redirect file descriptor 2 (diagnostic output)

SYNOPSIS

fd2 [fd2arg] command [command-arg] ...

DESCRIPTION

Fd2 executes *command* with file descriptor 2 (the diagnostic output) redirected to a file or to the standard output. There are three forms:

fd2 –file comd ...	[write on *file*]
fd2 ––file comd ...	[append to *file*]
fd2 + comd ...	[causes file descriptor 2 to be made the same as file descriptor 1]

In either of the first two cases, omission of *file* causes **msg.out** to be used as the output file. Omission of *fd2arg* has the effect of **–msg.out.**

NAME

fgrep – search a file for lines containing keywords

SYNOPSIS

fgrep [**–b**] [**–c**] [**–e**][**–f**] [**–n**] [**–v**] pattern [file] ...

DESCRIPTION

fgrep searches the input files (standard input default) for all lines containing one or more keywords denoted by *pattern*. Normally, each containing line is copied to the standard output. The **bcfnv** flags modify the normal behavior as in *egrep*(1). The **–e** flag causes a match to occur if and only if a keyword matches an input line exactly. Without the **–f** flag, *pattern* can be only a single keyword. With the **–f** flag, *pattern* is the name of a file containing a sequence of keywords terminated by newlines. The keywords in this file then constitute the search pattern. A keyword is any string of characters except '\0' and newline. No metacharacters are assumed.

SEE ALSO

egrep(1)

BUGS

Lines longer than 512 characters are not printed completely.

NAME

file – determine file type

SYNOPSIS

file file ...

DESCRIPTION

File 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.

NAME

find – find files

SYNOPSIS

find pathname-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each pathname in the *pathname-list* (i.e., one or more pathnames) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, −*n* means less than *n* and *n* means exactly *n*.

−**name** filename True if the *filename* argument matches the current file name. Normal Shell argument syntax may be used if escaped (watch out for '[', '?' and '*').

−**perm** onum True if the file permission flags exactly match the octal number *onum* (see *chmod*(I)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(II)) become significant and the flags are compared: *(flags&onum)==onum*.

−**type** c True if the type of the file is *c*, where *c* is **b**, **c**, **d** or **f** for block special file, character special file, directory or plain file.

−**links** n True if the file has *n* links.

−**user** uname True if the file belongs to the user *uname*.

−**group** gname As it is for −**user** so shall it be for −**group** (someday).

−**size** n True if the file is *n* blocks long (512 bytes per block).

−**atime** n True if the file has been accessed in *n* days.

−**mtime** n True if the file has been modified in *n* days.

−**exec** command True if the executed command returns a zero value as exit status. The end of the command must be punctuated by an escaped semicolon. A command argument '{}' is replaced by the current pathname.

−**ok** command Like −**exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds **y**.

−**print** Always true; causes the current pathname to be printed.

The primaries may be combined with these operators (ordered by precedence):

! Prefix *not*.

−**a** Infix *and*, second operand evaluated only if first is true.

−**o** Infix *or*, second operand evaluated only if first is false.

(expression) Parentheses for grouping. (Must be escaped.)

To remove all files named 'a.out' or '*.o' that have not been accessed for a week:

```
find / "(" -name a.out -o -name "*.o" ")" -a -atime +7 -a -exec rm {} ";"
```

FILES

/etc/passwd

SEE ALSO

sh(I), if(I), fs(V)

BUGS

Test (see *if*(I)) can be useful with *find*. However, since *test* is implemented within the Shell, you must use something like:

```
-exec sh -c "test args" ";"
```

NAME

gath – gather real and virtual files

SYNOPSIS

gath [**–ih**] file ...

DESCRIPTION

Gath concatenates the named files and writes them to standard output. Tabs are expanded into spaces according to the format specification for each file (see *fspec*(V)). The size limit and margin parameters of a format specification are also respected. Non-graphic characters other than tabs are identified by a diagnostic message and excised. The output of *gath* contains no tabs unless the **–h** flag is set, in which case it is written with standard tabs (every eighth column).

Any line of any of the files which begins with '~' is interpreted by *gath* as a control line. A line beginning '~ ' (tilde,space) specifies a sequence of files to be included at that point. A line beginning '~!' specifies a UNIX command; that command is executed, and its output replaces the '~!' line in the *gath* output.

Setting the **–i** flag prevents control lines from being interpreted and causes them to be output literally.

A file name of '–' at any point refers to standard input, and a control line consisting of '~.' is a software end-of-file. Keywords may be defined by specifying a replacement string which is to be substituted for each occurrence of the keyword. Input may be collected directly from the terminal, with several alternatives for prompting.

In fact, all of the special arguments and flags recognized by the *send* command are also recognized and treated identically by *gath*. Several of them are only useful, however, in the context where an RJE job is being submitted. The same program implements the two commands, so *gath* has a potential which is not apparent from its name. Refer to the description of *send* for definitive information about *gath*.

SEE ALSO

send(I), fspec(V)

NAME

get – get generation from SCCS file

SYNOPSIS

get [**–r**rel[.lev[.br[.seq]]]] [**–c**cutoff] [**–i**incl-list] [**–x**excl-list] [**–a**serial] [**–k**] [**–e**] [**–l**[**p**]] [**–p**] [**–m**] [**–n**] [**–s**] [**–b**] [**–g**] [**–t**] name ...

DESCRIPTION

Get generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with "–". The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file*. See **FILES**, below, for an explanation of how the name of this file is determined.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

–r The SCCS identification string (SID) of the change level to be generated. If the entire argument is omitted, the meaning is the same as if the default SID were specified (see *admin*(I)). If there is no default SID in the SCCS file the highest release which has deltas is used. If only the release is specified, the level defaults to the highest level in that release. A release and level completely identifies a specific change level. If a branch is also specified and the sequence is omitted, the sequence defaults to the highest sequence in the branch. A release, level, branch, and sequence also completely identifies a specific change level. (All deltas are identified either by a 2 component SID—release and level, or by a 4 component SID—release, level, branch, and sequence. SID's with 4 components identify deltas which have heretofore been called "non-propagating".)

–c Cutoff date-time, in the form YY[MM[DD[HH[MM[SS]]]]]. No delta which was created after the specified cutoff date-time will be applied. Units omitted from the date-time default to their maximum possible values; that is, "–c7502" is equivalent to "–c750228235959". Any number of non-numeric characters may separate the various 2 digit pieces of the cutoff date-time. This feature allows one to specify a cutoff date in the form: ""–c77/2/2 9:22:25"". Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested *gets* within, say the input to a *send*(I) command:

```
~!get "–c%E% %U%" s.file
```

–i This argument is used to specify a list of deltas to be included (forced to be applied). The list has the following syntax:

```
<list>   ::= <range>
           | <list> , <range>
<range>  ::= <delta>
```

```
                | <delta> - <delta>
<delta>  ::=    <rel>
                | <rel> . <lev>
                | <rel> . <lev> . <br>
                | <rel> . <lev> . <br> . <seq>
```

If a level is omitted from a delta specification the highest level of the specified release is assumed. If a branch is specified, but the sequence is omitted the highest sequence of the specified branch is assumed.

-x This argument is similar to **i** except that it is followed by a list of deltas to be excluded (forced to not be applied).

-a The serial number of the change level to be generated (see *sccsfile*(V)). This keyletter is used by the *comb*(I) command; it is not a generally useful keyletter, and most users will probably never use it. If both the **r** and **a** keyletters are specified, the **a** keyletter is used. Care should be taken when using the **a** keyletter in conjunction with the **e** keyletter, as the SID of the delta to be created may not be what one expects. The **r** keyletter can be used with the **a** and **e** keyletters to control the naming of the SID of the delta to be created.

-k This argument suppresses replacement of identification keywords (see below) by specific values. The **k** argument is implied by the **i**, **x** or **e** arguments.

-e This argument indicates that this *get* is for the purpose of making a delta with a later execution of *delta*. It causes creation, or updating of a *p-file* (see FILES). Another *get* with an **e** argument, if at the same delta or for the same new SID, may not be executed until the delta is made. If the *g-file* generated by a *get* with an **e** argument is ruined, a new one may be obtained by executing another *get* with a **k** argument instead of an **e**. Note that although the **c** argument may be used in combination with **e**, *delta* will not use it when regenerating the *g-file* for the purpose of determining what changed. When the **e** argument is supplied the protection restrictions determined by the ceiling, the floor, and the list of users authorized to make deltas are enforced.

-l This argument causes a delta summary to be written into an *l-file* (see FILES). If **-lp** is used then an *l-file* is not created; the delta summary is written on the standard output instead. The *reform*(I) command can be used to truncate lines of the *l-file*.

-p This argument causes the generated text to be written to the standard output instead of to a *g-file*. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the **s** argument is supplied, in which case it disappears.

-s This argument suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

-m This argument causes each generated text line to be preceded by the SID of the delta which inserted that text line. The format is: SID, followed by a horizontal tab, followed by the text line.

-n This argument causes each generated text line to be preceded with the %M% identification keyword (see below). The format is: %M% identification keyword, followed by a horizontal tab, followed by the text line. When both the **m** and **n**

arguments are supplied the format is: %M% identification keyword, followed by a horizontal tab, followed by the **m** argument format.

—b This argument is used with the **e** argument to indicate that the new delta should have an SID in a new branch. This argument is allowed only if the **b** flag exists in the file; see *admin*(I).

—g The **g** argument suppresses the actual getting of source. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

—t The **t** argument is used to access the most recent ("top") delta in a given release (i.e., when no **r** argument is supplied, or an argument of the form **r**rel is supplied).

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a newline) before it is processed. If the **i** argument is supplied included deltas are listed following the notation "Included"; if the **x** argument is supplied excluded deltas are listed following the notation "Excluded".

Identifying information is inserted into the generated text by replacing *identification keywords* by appropriate values, wherever they occur. The following keywords are available:

Keyword	*Value*
%M%	Module name; either the value of the **m** flag in the file (see *admin*(I)), or the *g-file* name—see **FILES**.
%I%	SCCS identification string (SID)—**%R%.%L%.%B%.%S%**.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%E%	Date of newest applied delta (YY/MM/DD).
%G%	Date of newest applied delta (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%U%	Time of newest applied delta (HH:MM:SS).
%Y%	The value of the **t** flag in the file (see *admin*(I)).
%F%	File name.
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4 characters @(#) (used to construct strings recognizable by *what*(I)).
%W%	A shorthand notation for constructing *what*(I) strings for UNIX program files. **%W% = %Z%%M%**<horizontal-tab>**%I%**
%A%	Another shorthand notation for constructing *what*(I) strings for non-UNIX program files. **%A% = %Z%%Y% %M% %I%%Z%**

FILES

Several auxiliary files may be created by *get*. These files are known generically as the *g-file*, *l-file*,

p-file, and *z-file*. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form "**s.***modulename*", the auxiliary files are named by replacing the leading "s" with the tag. The *g-file* is an exception to this scheme: the *g-file* is named by removing the "s.". For example, if the SCCS file name is "s.xyz.c", the auxiliary file names would be "xyz.c", "l.xyz.c", "p.xyz.c", and "z.xyz.c", respectively.

The *g-file*, which contains the generated text, is created in the current directory (unless the **p** argument is supplied, or zero lines of text were generated). It is owned by the real user. If the **k** argument is supplied or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* is also created (unless a **p** follows the **–l**) in the current directory, if the **l** argument is supplied; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory. The *l-file* contains a table showing which deltas were applied. The following is printed for each delta in the SCCS file:

a) Blank if the delta was applied; "*" otherwise.
b) Blank if the delta was applied or wasn't applied and ignored; "*" if the delta wasn't applied and wasn't ignored.
c) A code indicating a "special" reason why the delta was or was not applied:
   "I": Included.
   "X": Excluded.
   "C": Cut off (by a **c** argument).
d) Blank.
e) SCCS identification string (SID).
f) Tab character.
g) Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h) Blank.
i) Creator.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an **e** argument along to *delta*. Its contents are used to prevent a subsequent execution of *get* with an **e** argument until *delta* is executed (subject to the conditions described above under the **e** keyletter description). The *p-file* is created in the directory containing the SCCS file (which might, of course, also be the current directory), and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID this delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time of the get (*not* the cutoff date-time), followed by a blank and the **–i** keyletter argument if it was present, followed by a blank and the **–x** keyletter argument if it was present, followed by a newline. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same gotten SID or the same new SID.

The *z-file* is created in the directory containing the SCCS file for the duration of updating the *p-file*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444. It serves as a *lock-out* mechanism against simultaneous updates. Its contents are (in binary; 2 bytes) the process ID of the command (i.e., *get*) that created it.

SEE ALSO

admin(I), delta(I), prt(I), what(I), help(I), sccsfile(V),
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

BUGS

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the e argument is supplied.

NAME

goto – command transfer

SYNOPSIS

goto label

DESCRIPTION

Goto is allowed only when the Shell is taking commands from a file. The file is searched from the beginning for a line beginning with ':' followed by one or more spaces followed by the *label*. If such a line is found, the *goto* command returns. Since the read pointer in the command file points to the line after the label, the effect is to cause the Shell to transfer to the labelled line.

':' is a do-nothing command that is ignored by the Shell and only serves to place a label.

SEE ALSO

sh(I)

NAME

graph – draw a graph

SYNOPSIS

graph [option] ... | plotter

DESCRIPTION

Graph with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. The graph is written on the standard output to be piped to the *plotter* program for a particular device; see *plot*(I).

If the coordinates of a point are followed by a nonnumeric string, that string is printed as a label beginning at the point. Labels may be surrounded with quotes "...", in which case they may contain blanks or begin with numeric characters; 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, or is assumed to be 1 if next argument is not a number. A second optional argument is the starting point for the automatic abscissa.

−c Character string given by next argument is default label for each point.

−d Omit connections between points. (Disconnect.)

−g*n* Grid style:
n=0, no grid
n=1, axes only
n=2, complete grid (default).

−l Next argument is label for graph.

−s Save screen, don't erase before plotting.

−x Next 1 (or 2) arguments are lower (and upper) *x* (abcissa) limits. Third argument, if present, is grid spacing on *x* axis. Normally these quantities are determined automatically.

−y Similarly for *y* (ordinate) axis.

−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.

Points are connected by straight line segments in the order they appear in input. If a specified lower limit exceeds the upper limit, or if the automatic increment is negative, the graph is plotted upside down. Automatic abscissas begin with the lower *x* limit, or with 0 if no limit is specified. Labels are placed so that the center of an initial letter such as + will fall approximately on the plotting point.

SEE ALSO

plot(I), spline(I)

BUGS

Graph stores all points internally even when limits are explicit, so utterly enormous graphs can fail unnecessarily.

NAME

grep – search a file for a pattern

SYNOPSIS

grep [**–v**] [**–b**] [**–c**] [**–n**] [**–s**] expression [file] ...

DESCRIPTION

Grep searches the input files (standard input default) for lines matching the regular expression. Normally, each line found is copied to the standard output. If the **–v** flag is used, all lines but those matching are printed. If the **–c** flag is used, only a count of matching lines is printed. If the **–n** flag is used, each line is preceded by its relative line number in the file. If the **–b** flag is used, each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

The **–s** flag suppresses the error messages that *grep* would otherwise give for non-existent (or unreadable) files.

In all cases the file name is shown if there is more than one input file.

For a complete description of the regular expression, see *ed*(I). Care should be taken when using the characters $ * [^ | ( ) and \ in the *expression*, as they are also meaningful to the Shell. It is generally necessary to enclose the entire *expression* argument in quotes.

SEE ALSO

ed(I), egrep(I), fgrep(I), rgrep(I), sed(I), sh(I)

BUGS

Lines are limited to 256 characters; longer lines are truncated.
Unfortunately, *grep* does not recognize all of the regular expression operators that *ed*(I) does.

NAME

gsi – handle special functions of GSI300 terminal

SYNOPSIS

gsi [**+12**] [**−***n*] [**−d***t,l,c*]

DESCRIPTION

Gsi supports special functions, and optimizes the use, of the GSI300 (DASI300 or DTC300) terminal. 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. It permits convenient use of 12-pitch text. It also reduces printing time (5 to 70%). *Gsi* can be used to print equations neatly, in the sequence:

neqn file ... | nroff | gsi

WARNING: if your terminal has a PLOT switch, make sure it is turned ON before *gsi* is used.

The behavior of *gsi* 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/inch text. GSI terminals normally allow only two combinations: 10-pitch, 6 lines/inch, or 12-pitch, 8 lines/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 4 vertical plot increments. Because each increment equals 1/48 of an inch, a 10-pitch line-feed requires 8 increments, while a 12-pitch line-feed needs only 6. The first digit of *n* overrides the default value, thus allowing for individual taste in the appearance of subscripts and superscripts. For example, *nroff(1)* half-lines could be made to act as quarter-lines by using **−2**. The user could also obtain appropriate half-lines for 12-pitch, 8 lines/inch mode by using the option **−3** alone, having set the PITCH switch to 12-pitch.

−d*t,l,c* controls delay factors. The default setting is **−d**3,90,30. GSI 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)* requests 2 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 **−d**3,30,5. The value **−d**0,1 is a good one to use for C programs that have many levels of indentation.

Note that the delay control interacts heavily with the carriage return and line feed delays being used at the time: see *GSI300(VII)*. The *stty(1)* modes **nl0 cr2** or **nl0 cr3** are recommended for most uses.

NOTE: *gsi* always synchronizes its buffering so that it 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 any response.

In many cases, the following sequences are equivalent:

nroff −T300 files ... and nroff files ... | gsi
nroff −T300−12 files ... and nroff files ... | gsi +12

The use of *gsi* can thus often be avoided unless special delays or options are required.

Here are the *neqn(I)* names and resulting output for the special characters supported:

Name	Symbol	Name	Symbol
alpha	α	OMEGA	Ω
beta	β	partial	∂
delta	δ	phi	ϕ
DELTA	Δ	PHI	Φ
epsilon	ϵ	psi	ψ
eta	η	PSI	Ψ
gamma	\	pi	π
GAMMA	Γ	PI	Π
infinity	∞	rho	ρ
integral	$\int$	sigma	σ
lambda	λ	SIGMA	Σ
LAMBDA	Λ	tau	τ
mu	μ	theta	θ
nabla(del)	∇	THETA	Θ
not	—	xi	ξ
nu	ν	zeta	ζ
omega	ω		

SEE ALSO

450(I), graph(I), greek(V), GSI300(VII), mesg(I), neqn(I), plot(I), stty(I), tabs(I)

BUGS

Some characters in the above table can't be correctly printed in column 1 because the print head cannot be moved to the left from there. If your output contains much Greek and/or reverse line feeds, use friction feed instead of a forms tractor. Although good enough for drafts, the latter has a tendency to slip when reversing direction, distorting Greek characters, and misaligning the first line after a long set of reverse line feeds.

Gsi is definitely *not* usable with the "second generation" models of the GSI300, such as the GSI300S or DASI450.

NAME

help – ask for help

SYNOPSIS

help [arg] ...

DESCRIPTION

Help finds information to explain a message from a command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names, of one of the following types:

type 1 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 (e.g., "ge6", for message 6 from the *get* command).

type 2 Does not contain numerics (as a command, such as *get*)

type 3 Is all numeric (e.g., "212")

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try "help stuck".

FILES

The ASCII file searched for the explanatory information for each type of argument is as follows:

type 1 /usr/lib/help/*prefix-of-argument*

type 2 /usr/lib/help/cmds

type 3 /usr/lib/sccs.hf

If the file to be searched for either a type 1 or a type 2 argument does not exist, the search will be attempted on the file for the type 3 argument. In no case, however, will more than one file be searched per argument.

Anyone wishing to modify the files should list out portions of them – the format will be obvious.

DIAGNOSTICS

Use *help* for help.

NAME

hp – handle special functions of HP 2640 terminal

SYNOPSIS

hp [–e] [–m]

DESCRIPTION

Hp supports special functions of the Hewlett-Packard 2640 family of terminals, with the primary purpose of producing accurate representations of most *nroff*(I) output. Typical uses are:

nroff –h files ... | hp or: nroff –h –s files ... | hp

In the latter case, *nroff* will stop at the beginning of each page including the first and wait for you to hit LINE FEED to initiate output. Regardless of the hardware options on your terminal, *hp* does 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 option, you can see Greek and other special characters.

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 Underline 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 scientific characters, *hp* provides the same set as does *gsi*(I), except that "not" is approximated by a right arrow, and only the top half of the integral sign is shown. The display is adequate for examining output from *neqn*(I).

DIAGNOSTICS

"line too long" if representation of a line exceeds 300 characters, which would occur, for instance, if you underlined every other character in an 80-character line containing many Greek characters.

EXIT CODES

0 – normal
1 – for any error

SEE ALSO

gsi(I), HP2640(VII), neqn(I), nroff(I)

BUGS

Note that the second or later characters in an overstriking sequence are always assumed to be underlines. For example, a bullet made from lower-case "o" overstruck with "+" appears as an "o" that is either underlined or shown in Inverse Video. Although some terminals do provide numerical superscript characters, no effort is made to display them. The programming is ugly, and most terminals do not possess this feature.

NAME

if − conditional command

SYNOPSIS

if expr command [arg ...]

if expr **then**
command(s)
...
[**else** [command]
...]
endif

test expr

DESCRIPTION

If evaluates the expression *expr*. In the first form above, if *expr* is true, the given *command* is executed with the given arguments. The command may be another *if*.

In the second form, if *expr* is true, the commands between the *then* and the next unmatched *else* or *endif* are executed. If *expr* is false, the commands after *then* are skipped, and the commands after the optional *else* are executed. Zero or one commands may be written on the same line as the *else*. In particular, *if* may be used this way. The pseudo commands *else* and *endif* (whichever occurs first) must not be hidden behind semicolons or other commands. This form may be nested: every *then* needs a matching *endif*.

Test is an entry to *if* that evaluates the expression and returns exit code 0 if it is true, and code 1 if it is false or in error.

The following primitives are used to construct the *expr:*

−r file — true if the file exists and is readable.

−w file — true if the file exists and is writable.

−s file — true if the file exists and has a size greater than zero.

−f file — true if the file exists and is an ordinary file.

−d file — true if the file exists and is a directory.

−z s1 — true if the length of string *s1* is zero.

−n s1 — true if the length of string *s1* is nonzero.

s1 = s2 — true if the strings *s1* and *s2* are equal.

s1 != s2 — true if the strings *s1* and *s2* are not equal.

n1 **−eq** n2
n1 **−ne** n2
n1 **−gt** n2
n1 **−ge** n2
n1 **−lt** n2
n1 **−le** n2 — true if the stated algebraic relationship exists. The arguments *n1* and *n2* must be integers.

{ command } The bracketed command is executed to obtain the exit status. Status zero is considered *true*. The command must **not** be another *if*.

These primaries may be combined with the following operators:

!	unary negation operator
−a	binary *and* operator
−o	binary *or* operator
(expr)	parentheses for grouping.

−a has higher precedence than **−o**. Notice that all the operators and flags are separate arguments to *if* and hence must be surrounded by spaces. Notice also that parentheses are meaningful to the Shell and must be escaped.

EXIT CODES

0 − true expression, no error.
1 − false condition or error.

SEE ALSO

exit(I), goto(I), sh(I), switch(I), while(I), exit(II)

DIAGNOSTICS

if:missing endif
if:syntax error: value
if:non-numeric arg: value
if:no command: name
else:missing endif

Test may issue any of the *if* messages above, except the first.

BUGS

In general, *if, else, endif,* and *test* must not be hidden behind semicolons on a command line. Many of the effects are obtained by searching the input file and adjusting the read pointer appropriately. Thus, including any of these commands in a part of the file intended to be read by a command other than the shell may cause strange results if they are encountered while searching. These commands ignore redirection or piping of their standard input or output. Commands executed by *if* or *test* may be affected by redirections, but this practice is undesirable and should be avoided.

NAME

kill – terminate a process

SYNOPSIS

kill [–signo] processid ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. This will normally terminate the process, unless it is caught. The process number of each asynchronous process started with '&' is reported by the Shell. Process numbers can also be found by using *ps*(I).

The details of the kill are described in *kill*(II). For example, if process number 0 is specified, all processes in the process group are signaled.

If a signal number preceded by "–" is given as the first argument, that signal is sent instead. For example, **–9** will guarantee a kill.

SEE ALSO

ps(I), sh(I), kill(II), signal(II)

NAME

ld – link editor

SYNOPSIS

ld [**−sulxrdni**] [**−o** name] file ...

DESCRIPTION

Ld combines several object programs into one; resolves external references; and searches libraries. In the simplest case several object *files* are given, and *ld* combines them, producing an object module which can be either executed or become the input for a further *ld* run. (In the latter case, the **−r** option must be given to preserve the relocation bits.) The output of *ld* is left on **a.out.** This file is made executable only if no errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output is the function named **main.**

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries is important.

Ld understands several flag arguments which are written preceded by a '−'. Except for **−l**, they should appear before the file names.

−s 'Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by *strip*(I).

−u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

−l This option is an abbreviation for a library name. **−l** alone stands for '/lib/liba.a', which is the standard system library for assembly language programs. **−l***x* stands for '/lib/lib*x*.a', where *x* can be a string. If that does not exist, *ld* tries '/usr/lib/lib*x*.a'. A library is searched when its name is encountered, so the placement of a **−l** is significant.

−x Do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

−X Save local symbols except for those whose names begin with 'L'. This option is used by *cc* to discard internally generated labels while retaining symbols local to routines.

−r Generate relocation bits in the output file so that it can be the subject of another *ld* run. This flag also prevents final definitions from being given to common symbols, and suppresses the 'undefined symbol' diagnostics.

−d Force definition of common storage even if the **−r** flag is present.

−n Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up to the first possible 4K word boundary following the end of the text.

−i When the output file is executed, the program text and data areas will live in separate address spaces. The only difference between this option and **−n** is that here the data starts at location 0.

-o The *name* argument after **-o** is used as the name of the *ld* output file, instead of **a.out.**

FILES

/lib/lib?.a	libraries
/usr/lib/lib?.a	more libraries
a.out	output file

SEE ALSO

as(I), ar(I), cc(I)

NAME

lex – generate programs for simple lexical tasks

SYNOPSIS

lex [–[**rctvfn**]] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analysis of text.

The input file(s) contain strings and expressions to be searched for, and C text to be executed when found. A file "lex.yy.c" is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in yytext, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in

```
[abx–z]
```

to indicate a, b, x, y, and z; and the operators *, +, and ? mean respectively any non-negative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character '.' is the class of all ASCII characters except newline. Parentheses for grouping and vertical bar for alternation are also supported. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character % at the end of an expression requires a trailing newline. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in yytext, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus

```
[a–zA–Z]+
```

matches a string of letters.

Three subroutines defined as macros are expected: input() to read a character; unput(c) to replace a character read; and output(c) to place an output character. They are defined in terms of the standard streams (and the –lS standard I/O library), but you can override them. The program generated is named yylex(), and the library contains a main() which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function yymore() accumulates additional characters into the same yytext; and the function yyless(p) pushes back the portion of the string matched beginning at p, which should be between yytext and yytext+yyleng. The macros input and output use files "yyin" and "yyout" to read from and write to, defaulted to "stdin" and "stdout", respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the "lex.yy.c" file. All rules should follow a %%, as in YACC. Lines preceding %% which begin with a non–blank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done. Example:

```
D       [0–9]
%%
if      printf("IF statement\n");
[a–z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
```

```
switch (input())
        {
        case '/': break;
        case '*': unput('*');
        default: go to loop;
        }
}
```

The external names generated by *lex* all begin with the prefix "yy" or "YY".

The flags must appear before any files. The flag −r indicates Ratfor actions, −c indicates C actions and is the default, −t causes the "lex.yy.c" program to be written instead to standard output, −v provides a one-line summary of statistics of the machine generated, −f indicates "faster" compilation, so no packing is done, but it can handle much smaller machines only, −n will not print out the − summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

This is intended to replace the older version of Lex. The new standard I/O library is used, so actions must use it, and an "include" statement is automatically provided. A definition in the definitions section may refer to other definitions (but not to itself). The "%+" option has been eliminated. The notation r{d,e} in a rule indicates between d and e instances of regular expression r. It has higher precedence then '|', but lower than '*', '?', '+', and concatenation.

In the definitions section,

%p num sets the max. − of positions to num (dft = 2000)
%n num sets the max. − of states to num (dft = 500)
%t num sets the max. − of parse tree nodes to num (dft = 1000)
%a num sets the max. − of transitions to num (dft = 3000)

The use of one or more of the above automatically implies the −v option, unless the −n option is used.

SEE ALSO

yacc(I)
LEX − Lexical Analyzer Generator by M. E. Lesk and E. Schmidt.

BUGS

The Ratfor option is not yet fully operational.

NAME

ln – make a link

SYNOPSIS

ln name1 [name2]

DESCRIPTION

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, etc.) may have several links to it. There is no way to distinguish a link to a file from its original directory entry; any changes in the file are effective independently of the name by which the file is known.

Ln creates a link to an existing file *name1*. If *name2* is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of *name1*.

It is forbidden to link to a directory or to link across file systems.

SEE ALSO

rm(I)

BUGS

There is nothing particularly wrong with *ln*, but *tp* doesn't understand about links and makes one copy for each name by which a file is known; thus if the tape is extracted several copies are restored and the information that links were involved is lost.

NAME

login — sign onto UNIX

SYNOPSIS

login [username]

DESCRIPTION

The *login* command is used when a user initially signs onto UNIX, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. See 'How to Get Started' for how to dial up initially.

If *login* is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it will not appear on the written record of the session.

After a successful login, accounting files are updated and the user is informed of the existence of *.mail* and message-of-the-day files. *Login* initializes the user and group IDs and the working directory, then executes a command interpreter (usually *sh*(I)) according to specifications found in a password file.

Login is recognized by the Shell and executed directly (without forking).

FILES

/etc/utmp	accounting
/usr/adm/wtmp	accounting
.mail	mail
/etc/motd	message-of-the-day
/etc/passwd	password file

SEE ALSO

init(VIII), getty(VIII), mail(I), passwd(I), passwd(V), sh(I), su(I)

DIAGNOSTICS

'login incorrect,' if the name or the password is bad. 'No Shell', 'cannot open password file,' 'no directory': consult a UNIX programming counselor. System hangs up a line left in login state.

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff old-spec new-spec [pr-args] sccsfile ...

DESCRIPTION

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. The *old-spec* is any valid *get*(I) specifier (e.g., –r1.1) for the old version to be gotten. Similarly, *new-spec* is any valid *get*(I) specifier (e.g., –r1.4) for the new version to be gotten. The *pr-args* are any valid *pr*(I) arguments which begin with a "–", except for "–h" (the output of *sccsdiff* is piped through *pr*(I)). Any number of SCCS files may be specified, but the *old-spec* and *new-spec* apply to all files.

Sccsdiff is a simple shell procedure; interested persons should "cat /usr/bin/sccsdiff" to discover how it works.

FILES

/tmp/get?????	temporary for old gotten version
/usr/bin/bdiff	program that generates differences

SEE ALSO

get(I), help(I), pr(I), bdiff(I)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

logname, logdir, logtty – information from login

SYNOPSIS

logname
logdir
logtty

DESCRIPTION

Logname prints the user's login name.

Logdir prints the login directory pathname.

Logtty prints the single character tty letter (and never prints 'x').

These data are created by *login*(I).

NAME

ls – list contents of directory

SYNOPSIS

ls [**−ltasdruifg**] name ...

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents. There are several options:

−l list in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

−t sort by time modified (latest first) instead of by name, as is normal

−a list all entries; usually those beginning with '.' are suppressed

−s give size in blocks for each entry

−d if argument is a directory, list only its name, not its contents (mostly used with **−l** to get status on directory)

−r reverse the order of sort to get reverse alphabetic or oldest first as appropriate

−u use time of last access instead of last modification for sorting (**−t**) or printing (**−l**)

−i print i-number in first column of the report for each file listed

−f force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **−l**, **−t**, **−s**, and **−r**, and turns on **−a**; the order is the order in which entries appear in the directory.

−g Give group ID instead of owner ID in long listing.

The mode printed under the **−l** option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
− if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable
w if the file is writable
x if the file is executable
− if the indicated permission is not granted

The group-execute permission character is given as s if the file has set-group-ID mode; likewise the user-execute permission character is given as s if the file has set-user-ID mode.

The last character of the mode is normally blank but is printed as "t" if the 1000 bit of the mode is on. See *chmod*(I) for the current meaning of this mode.

FILES

/etc/passwd to get user ID's for **ls −l**.

NAME

m4 – macro processor

SYNOPSIS

m4 [files]

DESCRIPTION

M4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no arguments, or if an argument is '–', the standard input is read. The processed text is written on the standard output.

Macro calls have the form

name(arg1,arg2, . . . , argn)

The '(' must immediately follow the name of the macro. If a defined macro name is not followed by a '(', it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Potential macro names consist of alphabetic letters, digits, and underscore '_', where the first character is not a digit.

Left and right single quotes (`') are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string.

undefine removes the definition of the macro named in its argument.

ifdef If the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null.

changequote Change quote characters to the first and second arguments. *Changequote* without arguments restores the original values (i.e., `').

divert *M4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum returns the value of the current output stream.

dnl reads and discards characters up to and including the next newline.

ifelse has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

eval evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, −, *, /, %, ^ (exponentiation); relationals; parentheses.

len returns the number of characters in its argument.

index returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.

substr returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

include returns the contents of the file named in the argument.

sinclude is identical to *include*, except that it says nothing if the file is inaccessible.

syscmd executes the UNIX command given in the first argument.

errprint prints its argument on the diagnostic output file.

dumpdef prints current names and definitions, for the named items, or for all if no arguments are given.

SEE ALSO

The M4 Macro Processor by B. W. Kernighan and D. M. Ritchie.

NAME

mail – send mail to designated users

SYNOPSIS

mail [**–yn**] [person ...]
mail **–f** file

DESCRIPTION

Mail with no argument searches for a file called **.mail**, prints it in reverse chronological order if it is nonempty, then asks if it should be saved. If the answer is **y**, the mail is added to **mbox**. In either case, **.mail** is truncated to zero length. To leave **.mail** untouched, hit 'delete.' The question can be answered on the command line with the argument **–y** or **–n**.

Mail tries to use **.mail** and **mbox** in the current directory. But if **.mail** doesn't exist, *mail* uses **.mail** and **mbox** in your *login* directory instead.

When *persons* are named, *mail* takes the standard input up to an end-of-file (or a line with just '.') and adds it to each *person's* **.mail** file. The message is preceded by the sender's name and a postmark. A *person* is a user name recognized by *login*(I). Mail is sent to the *login* directory of that user.

The **–f** option causes the named file to be printed as if it were mail.

When a user logs in he is informed of the presence of mail.

To receive mail, a **.mail** file must exist in your *login* directory, and it must be writable by everyone. However, it need not be readable by everyone.

FILES

/etc/passwd	to identify sender and locate persons
.mail	input mail
mbox	saved mail
/tmp/m?????	temp file

SEE ALSO

write (I)

NAME

make − make a program

SYNOPSIS

make [−**f** descfile] [−**p**] [−**i**] [−**s**] [−**r**] [−**n**] [−**t**] file ...

DESCRIPTION

Make may be used to mechanize program creation and maintenance, while ensuring that all constituents are current. A graph of dependencies is specified in the *descfile(s)*. The standard input will be read if − is given for *descfile*. If no −**f** options are present, the file named **makefile** or, if absent, the file named **Makefile** in the current directory is used. The −**p** option prints out a version of that graph. Each file name argument is 'made', as described below. If no such arguments are present, the initial node in the description file is made.

The *descfile* consists of a sequence of entries that describe the prerequisites and operations for creating an object (usually a file). The first line of each entry contains the names of the objects to be made, followed by a colon, optionally followed by a list of other files that must be available and current in order to remake it. Text following a semicolon on the first line, and all immediately following lines that begin with a tab, are fed to the Shell to make the object. Each line is fed to a separate instance of the Shell. All text following a sharp is taken to be a comment. For example:

```
pgm: x.o y.o ; cc x.o y.o -lp
        mv a.out pgm     # command to be done
x.o: dcls
```

Make walks the graph of dependencies. If a needed object depends on another that is not present or is younger than itself, it is remade. If an object's name ends in '.o', the description file, and then the current directory, are searched for a corresponding name ending in '.r', '.f', '.c', '.s', '.l' (Lex), '.y' (Yacc-C). (These default rules are not applied if the −**r** option is specified). If such a file is found and is younger than the object, a compilation command is executed. In the example above, if 'dcls' has been changed since 'pgm' was last made, 'x.c' will be recompiled and 'pgm' will be reloaded.

Each command line is printed before it is executed unless the −**s** option is specified on the command line or the special name '.SILENT' appears in the description. The command lines are printed but not executed if the −**n** option is specified. The date last modified is updated but the commands given are not executed for each file if the −**t** option is specified. (This option is useful when a source change is known to be incremental or benign).

Make examines the *exit*(II) status returned by each executed command. If the status is non-zero (i.e., if an error occurred), *make* aborts, unless either (a) the −**i** option was specified, or (b) the command name in the *descfile* was prefixed by '−'.

SEE ALSO

Make − A Program for Maintaining Computer Programs by S. I. Feldman.

DIAGNOSTICS

No description file argument
Cannot find description file
Syntax error
Don't know how to make xxx.

BUGS

Many UNIX commands return random status, which will cause *make* to assume that the command failed. In case of trouble, use the **-i** option or a minus sign on the command line.

NAME

man – print on-line documentation

SYNOPSIS

man [options] documents ...

DESCRIPTION

Man is a Shell command file that prints on-line documentation on the standard output by means of *nroff*(I) or *troff*(I). On-line documentation consists of manual pages from the PWB/UNIX User's Manual.

The command line to print manual pages consists of:

man [term] [**–s**] [section] title ...

where "term" is one of the following:

–t produces photocomposed output;

–g adapts the output to a DASI300 terminal in 12-pitch mode;

–450 adapts the output to the DASI450 terminal in 10-pitch mode;

–hp adapts the output to a Hewlett-Packard terminal;

–v followed by a space and a bin number, produces output on the Versatec printer. Exactly one bin number *must* be specified when the **–v** option is used.

The **–s** option prints only the SYNOPSIS portion of a manual page.

Section is the section number in the PWB/UNIX User's Manual in which a manual page is filed; it is specified as a single Arabic decimal digit. If *section* is omitted on the command line, the section number defaults to 1. If the page is not in the given section, then a search is made of *all* sections of the manual. If the page is not found (i.e., does not exist), an error message is produced.

Title is the name of the manual page. One or more titles may be specified in a single command.

Thus, the command line:

man –g 1 ed man tbl

would print out (in 12-pitch on a DASI300 terminal) the pages for the commands *ed*, *man*, and *tbl*, all of which can be found in Section I of the PWB/UNIX User's Manual.

DIAGNOSTICS

"man page for xxx not found" A manual page for xxx does not exist.

FILES

/usr/man/man[0-8] Installed PWB/UNIX pages.

SEE ALSO

Anyone who wishes to *write* manual pages like those accessed by this command should read *PWB/UNIX Manual Page Macros* by E. M. Piskorik.

NAME

mesg – permit or deny messages

SYNOPSIS

mesg [n] [y]

DESCRIPTION

Mesg with argument **n** forbids messages via *write*(I) by revoking non-user write permission on the user's terminal. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reverses the current permission. In all cases the previous state is reported.

FILES

/dev/tty?

SEE ALSO

write(I)

DIAGNOSTICS

'?' if the standard input file is not a terminal.

NAME

mkdir – make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 755. The standard entries '.' and '..' are made automatically.

SEE ALSO

rmdir(I)

NAME

mm – run off document with PWB/MM

SYNOPSIS

mm [options] [files]

DESCRIPTION

The *mm* command can be used to run off documents using *nroff*(I) and the PWB/MM text formatting codes. It has options to specify preprocessing by *tbl*(I) or by *neqn*(I) and for postprocessing by various output filters. The proper pipe sequences and the required arguments and flags for *nroff*(I) and PWB/MM are generated, depending on the options selected. For example, inclusion of the **–h** *nroff*(I) flag occurs unless *col*(I) is to be used or unless the **–450** option is specified.

The options for *mm* are listed below. Any other arguments or flags, e.g. **–rC3**, are passed to *nroff*(I) or to PWB/MM, as appropriate. The options can occur in any order, but they must appear before the files.

–e *neqn*(I) processing is needed.

–t *tbl*(I) is needed.

–c *col*(I) is needed.

–12 want 12 pitch mode. (Be sure that the 12-pitch switch is set on the terminal.)

–300 output onto a DASI 300 terminal. This is the *default* terminal type.

–hp output onto a HP 2640A.

–450 output onto a DASI 450.

–300S output onto a DASI 300S.

–300s output onto a DASI 300S.

–tn output onto a GE TermiNet 300.

–tn300 output onto a GE TermiNet 300.

–ti output onto a Texas Instrument terminal.

–37 output onto a TTY 37.

If several terminal types are specified, the last one takes precedence. Note that **–ti**, **–tn**, and **–tn300** all do the same thing; they all imply **–c**, and work for any terminal that lacks reverse line feed capability.

As an example,

```
mm –t –450 –rC3 –12 qqsv*
```

generates

```
tbl qqsv* | nroff –h –mm –rT1 –rC3 – | 450
```

If no arguments are given, *mm* prints a list of options.

If only options and unreadable files are specified, then *mm* terminates silently.

HINTS

1. *mm* may invoke *nroff*(I) with the **–h** flag. With this flag, *nroff*(I) assumes that the terminal has tabs set at every 8 character positions.

2. Use the **-o***list* option of *nroff*(1) to specify ranges of pages to be output.
3. When either **-t** or **-e** or both are specified and the **-o***list* does *not* cause the last page of the document to be printed, a "broken pipe" message from the Shell will result.

SEE ALSO

nroff(1)
PWBMM — Programmer's Workbench Memorandum Macros by D. W. Smith and J. R. Mashey.
Typing Documents with PWBMM by D. W. Smith and E. M. Piskorik.

NAME

mv – move or rename a file

SYNOPSIS

mv name1 name2

DESCRIPTION

Mv changes the name of *name1* to *name2*. If *name2* is a directory, *name1* is moved to that directory with its original file-name. Directories may only be moved within the same parent directory (just renamed).

If *name2* already exists, it is removed before *name1* is renamed. If *name2* has a mode which forbids writing, *mv* prints the mode and reads the standard input to obtain a line; if the line begins with y, the move takes place; if not, *mv* exits.

If *name2* would lie on a different file system, so that a simple rename is impossible, *mv* copies the file and deletes the original.

SEE ALSO

cp(I), cpio(I)

BUGS

It should take a **–f** flag, like *rm*, to suppress the question if the target exists and is not writable.

NAME

neqn – typeset mathematics on terminal

SYNOPSIS

neqn [file] ...

DESCRIPTION

Neqn is an *nroff*(I) preprocessor. The input language is the same as that of *eqn*(I). Normal usage is almost always:

neqn file ... | nroff

Output is meant for terminals with forward and reverse capabilities, such as the Model 37 TELETYPE® or GSI (DASI or DIABLO) terminals.

If no arguments are specified, *neqn* reads the standard input, so it may be used as a filter.

SEE ALSO

eqn(I), gsi(I), mm(I), DASI450(VII), GSI300(VII)
Typesetting Mathematics – User's Guide (2nd Edition) by B. W. Kernighan and L. L. Cherry.
New Graphic Symbols for EQN and NEQN by C. Scrocca.

BUGS

Because of some interactions with *nroff*(I), there may not always be enough space left before and after lines containing equations.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp group

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login*(I). The same person remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

A password is demanded if the group has a password and the user himself does not.

When most users log in, they are members of the group named 'other.'

FILES

/etc/group, /etc/passwd

SEE ALSO

login(I), group(V)

NAME

next — new standard input for shell procedure

SYNOPSIS

next [filename]

DESCRIPTION

This command causes *filename* to become standard input. The current input is never resumed. If no filename is given, the real terminal is assumed.

Next is executed within the shell.

SEE ALSO

sh(I)

NAME

nice – run a command at low priority

SYNOPSIS

nice [–number] command [arguments]

DESCRIPTION

Nice executes *command* with low scheduling priority. If the *number* argument is given, that priority (in the range 1–20) is used; if not, priority 4 is used.

The super-user may run commands with priority higher than normal by using a negative priority, e.g. '—10'.

SEE ALSO

nohup(I), nice(II)

NAME

nm – print name list

SYNOPSIS

nm [**–cnrupg**] [name]

DESCRIPTION

Nm prints the symbol table from the output file of a compiler or loader run. Each symbol name is preceded by its value (blanks if undefined) and one of the letters **U** (undefined) **A** (absolute) **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), or **C** (common symbol). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

If no file is given, the symbols in **a.out** are listed.

Options are:

–c list only C-style external symbols, that is those beginning with underscore '_'.

–g print only global (external) symbols

–n sort by value instead of by name

–p don't sort; print in symbol-table order

–r sort in reverse order

–u print only undefined symbols.

FILES

a.out

NAME

nohup – run a command immune to hangups

SYNOPSIS

nohup command [arguments]

DESCRIPTION

Nohup executes *command* with hangups, quits and interrupts all ignored. If output is not redirected by the user, it will be sent to */dev/null* (a "write-only" file).

SEE ALSO

nice(I), signal(II)

NAME

nroff, troff – text formatters

SYNOPSIS

nroff (or **troff**) [options] files

DESCRIPTION

NROFF and TROFF accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input. If no file names are given, input is taken from the standard input. The options, which may appear in any order so long as they appear before the filenames, are:

Option	*Effect*
–o*list*	Print only pages whose page numbers appear in *list*, which consists of numbers and number ranges separated by commas. A number range has the form *N*–*M* and means pages *N* through *M* inclusive; an initial –*N* means from the beginning to page *N*; and a final *N*– means from *N* to the end.
–n*N*	Number first generated page *N*.
–s*N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a new-line character. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow the changing of cassettes, and will resume after the phototypesetter START button is pressed.
–m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input files.
–r*aN*	Register *a* (one-character name) is set to *N*.
–i	Read standard input after the input files are exhausted.
–q	Invoke the simultaneous input-output mode of the **rd** request.
	NROFF Only
–T*type*	Specifies the output terminal type. Currently defined values for *type* are **37** for the (default) Model 37 TELETYPE®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300** for the DASI-300, **450** for the DASI-450 (or Diablo Hyterm) and **300S** for the DASI-300S. For 12-pitch, use **300-12**, **300S-12**, and **450-12**.
–e	Produce equally-spaced words in adjusted lines, using full terminal resolution.
–h	Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.
	TROFF Only
–t	Direct output to the standard output instead of the phototypesetter.
–f	Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-w Wait until phototypesetter is available, if it is currently busy.

-b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

-a Send a printable (ASCII) approximation of the results to the standard output.

-p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

-g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

FILES

/usr/lib/suftab suffix hyphenation tables
/tmp/ta00000 temporary file
/usr/lib/tmac.* standard macro files
/usr/lib/term/* (NROFF only) terminal driving tables
/usr/lib/font/* (TROFF only) font width tables

SEE ALSO

NROFF/TROFF User's Manual by J. F. Ossanna.
A TROFF Tutorial by B. W. Kernighan.
tbl(I).
For NROFF, see neqn(I), col(I), and tabs(I)
For TROFF, see eqn(I).

NAME

od – octal dump

SYNOPSIS

od [**–abcdho**] [file] [[**+**] offset[.][**b**]]

DESCRIPTION

Od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing, **–o** is default. The meanings of the format argument characters are:

a interprets words as PDP-11 instructions and dis-assembles the operation code. Unknown operation codes print as ???.

b interprets bytes in octal.

c interprets bytes in ascii. Unknown ascii characters are printed as \?.

d interprets words in decimal.

h interprets words in hex.

o interprets words in octal.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used. Thus *od* can be used as a filter.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If '.' is appended, the offset is interpreted in decimal. If '**b**' is appended, the offset is interpreted in blocks. (A block is 512 bytes.) If the file argument is omitted, the offset argument must be preceded by '**+**'.

Dumping continues until end-of-file.

SEE ALSO

db(I)

NAME

onintr – handle interrupts in shell files

SYNOPSIS

onintr [label]

DESCRIPTION

The *onintr* command catches interrupts received while the Shell is reading from a file. After the interrupt, and after any active process has completed, the Shell procedure is transferred to the label specified. Unless another *onintr* command is processed, the next interrupt will kill the Shell. The command without a label turns interrupts back on. The special case "onintr –" causes interrupts to be totally ignored, both by the Shell itself and by subsequent commands invoked by the Shell.

Onintr is executed within the Shell.

SEE ALSO

sh(I)

NAME

passwd – change login password

SYNOPSIS

passwd name password

DESCRIPTION

The *password* becomes associated with the given login *name*. This can only be done by the user who has that login name, or by the super-user. An explicit null argument ("") for the *password* argument removes any password.

FILES

/etc/passwd

SEE ALSO

login(I), passwd(V), crypt(III)

NAME

plot: t300, t300s, t450 – graphics filters

SYNOPSIS

t300
t300s
t450

DESCRIPTION

These commands read plotting instructions (see *plot*(V)) from the standard input, and produce device-dependent plotting instructions on the standard output.

T300 produces a plot for a GSI 300 terminal on the standard output.

T300s produces a plot for a GSI 300s terminal on the standard output.

T450 produces a plot for a DASI 450 terminal on the standard output.

SEE ALSO

graph(I), plot(III), plot(V)

NAME

pr – print file

SYNOPSIS

pr [**−h** header] [**−n**] [**+n**] [**−wn**] [**−ln**] [**−t**] [**−sc**] [**−m**] name ...

DESCRIPTION

Pr produces a printed listing of one or more files. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, *pr* prints its standard input, and is thus usable as a filter.

Options apply to all following files but may be reset between files:

−*n* produce *n*-column output

+*n* begin printing with page *n*

−h treat the next argument as a header to be used instead of the file name

−w*n* for purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72

−l*n* take the length of the page to be *n* lines instead of the default 66

−t do not print the 5-line header or the 5-line trailer normally supplied for each page

−s*c* separate columns by the single character *c* instead of by the appropriate amount of white space. A missing *c* is taken to be a tab.

−m print all files simultaneously, each in one column

Inter-terminal messages via *write*(I) are forbidden during a *pr*.

FILES

/dev/tty? to suspend messages.

SEE ALSO

cat(I), cp(I)

DIAGNOSTICS

none; files not found are ignored.

NAME

prof – display profile data

SYNOPSIS

prof [**–a**] [**–l**] [file]

DESCRIPTION

Prof interprets the file *mon.out* produced by the *monitor*(III) subroutine. Under default modes, the symbol table in the named object *file* (**a.out** default) is read and correlated with the **mon.out** profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **–a** option is used, all symbols are reported rather than just external symbols. If the **–l** option is used, the output is listed by symbol value rather than decreasing percentage.

In order for the number of calls to a routine to be tallied, the **–p** option of *cc* must have been given when the file containing the routine was compiled. This option also arranges for the **mon.out** file to be produced automatically.

FILES

mon.out	for profile
a.out	for namelist

SEE ALSO

monitor(III), profil(II), cc(I)

BUGS

Beware of quantization errors.

NAME

prt – print SCCS file

SYNOPSIS

prt [**–d**] [**–s**] [**–a**] [**–i**] [**–u**] [**–f**] [**–t**] [**–b**] [**–e**] [**–y**[SID]]
[**–c**[cutoff]] [**–r**[reverse-cutoff]] name ...

DESCRIPTION

Prt prints part or all of an SCCS file in a useful format. If a directory is named, *prt* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The keyletter arguments are as follows. Each is explained as though only one named file is to be processed, but the effects of any keyletter argument apply independently to each named file.

–d This keyletter normally causes the printing of delta table entries of the "D" type.

–s Causes only the first line of the delta table entries to be printed; that is, only up to the statistics. This keyletter is effective only if the **d** keyletter is also specified (or assumed).

–a Causes those types of deltas normally not printed by the **d** keyletter to be printed. These are types "R" (removed) and "H" (changed history). This keyletter is effective only if the **d** keyletter is also specified (or assumed).

–i Causes the printing of the serial numbers of those deltas included, excluded, and ignored. This keyletter is effective only if the **d** keyletter is also specified (or assumed).

The following format is used to print those portions of the SCCS file as specified by the above keyletters. The printing of each delta table entry is preceded by a newline character.

a) Type of delta ("D", "R", or "H").
b) Space.
c) SCCS identification string (SID).
d) Tab.
e) Date and time of creation.
(in the form YY/MM/DD HH:MM:SS)
f) Tab.
g) Creator.
h) Tab.
i) Serial number.
j) Tab.
k) Predecessor delta's serial number.
l) Tab.
m) Statistics.
(in the form inserted/deleted/unchanged)
n) Newline.

o) "Included:*tab*", followed by SID's of deltas included, followed by newline (only if there were any such deltas and if **i** keyletter was supplied).
p) "Excluded:*tab*", followed by SID's of deltas excluded, followed by newline (see note above).
q) "Ignored:*tab*", followed by SID's of deltas ignored, followed by newline (see note above).
r) "MRs:*tab*", followed by MR numbers related to the delta, followed by newline (only if any MR numbers were supplied).
s) Lines of comments (history), followed by newline (if any were supplied).

—u Causes the printing of the login-names of those users allowed to make deltas.

—f Causes the printing of the flags of the named file.

—t Causes the printing of the descriptive text contained in the file.

—b Causes the printing of the body of the SCCS file.

—e This keyletter implies the **d**, **i**, **u**, **f**, and **t** keyletters and is provided for convenience.

—y This keyletter will cause the printing of the delta table entries to stop when the delta just printed has the specified SID. If no delta in the table has the specified SID, the entire table is printed. If no SID is specified, the first delta in the delta table is printed. This keyletter will cause the entire delta table entry for each delta to be printed as a single line (the newlines in the normal multi-line format of the **d** keyletter are replaced by blanks) preceded by the name of the SCCS file being processed, followed by a ":", followed by a tab. This keyletter is effective only if the **d** keyletter is also specified (or assumed).

—c This keyletter will cause the printing of the delta table entries to stop if the delta about to be printed is older than the specified cutoff date-time (see *get*(I) for the format of date-time). If no date-time is supplied, the epoch 0000 GMT Jan. 1, 1970 is used. As with the **y** keyletter, this keyletter will cause the entire delta table entry to be printed as a single line and to be preceded by the name of the SCCS file being processed, followed by a ":", followed by a tab. This keyletter is effective only if the **d** keyletter is also specified (or assumed).

—r This keyletter will cause the printing of the delta table entries to begin when the delta about to be printed is older than or equal to the specified cutoff date-time (see *get*(I) for the format of date-time). If no date-time is supplied, the epoch 0000 GMT Jan. 1, 1970 is used. (In this case, nothing will be printed). As with the **y** keyletter, this keyletter will cause the entire delta table entry to be printed as a single line and to be preceded by the name of the SCCS file being processed, followed by a ":", followed by a tab. This keyletter is effective only if the **d** keyletter is also specified (or assumed).

If any keyletter but **y**, **c**, or **r** is supplied, the name of the file being processed (preceded by one newline and followed by two newlines) is printed before its contents.

If none of the **u**, **f**, **t**, or **b** keyletters is supplied, the **d** keyletter is assumed.

Note that the **s** and **i** keyletters, and the **c** and **r** keyletters are mutually exclusive; therefore, they may not be specified together on the same *prt* command.

The form of the delta table as produced by the y, c, and r keyletters makes it easy to sort multiple delta tables by time order. For example, the following will print the delta tables of all SCCS files in directory *sccs* in reverse chronological order:

```
prt -c sccs | grep . | sort '-rt<tab>' +2 -3
```

When both the y and c or the y and r keyletters are supplied, *prt* will stop printing when the first of the two conditions is met.

The *reform*(I) command can be used to truncate long lines.

See *admin*(I); *sccsfile*(V), and *SCCS/PWB User's Manual* for more information about the meaning of the output of *prt*.

SEE ALSO

admin(I), get(I), delta(I), what(I), help(I), sccsfile(V)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

ps — process status

SYNOPSIS

ps [**aklxt**] [namelist]

DESCRIPTION

Ps prints certain indicia about active processes. The **a** flag asks for information about all processes with terminals (ordinarily only one's own processes are displayed); **x** asks even about processes with no terminal; **l** asks for a long listing. The short listing contains the process ID, tty letter, the cumulative execution time of the process and an approximation to the command line. If the **k** flag is specified, the file */sys/sys/core* is used in place of */dev/mem*. This is used for postmortem system debugging. If a second argument is given, it is taken to be the file containing the system's namelist. If the **t** flag is used, the following character is taken to be the specific tty for which information is to be printed.

The long listing is columnar and contains

F — Flags associated with the process. 01: in core; 02: system process; 04: locked in core (e.g. for physical I/O); 10: being swapped; 20: being traced by another process.

S — The state of the process. 0: nonexistent; S: sleeping; W: waiting; R: running; I: intermediate; Z: terminated; T: stopped.

UID — The user ID of the process owner.

PID — The process ID of the process; as in certain cults it is possible to kill a process if you know its true name.

PPID — The process ID of the parent process.

CPU — Processor utilization for scheduling.

PRI — The priority of the process; high numbers mean low priority.

NICE — Used in priority computation.

ADDR — The core address of the process if resident, otherwise the disk address.

SZ — The size in blocks of the core image of the process.

WCHAN — The event for which the process is waiting or sleeping; if blank, the process is running.

TTY — The controlling tty for the process.

TIME — The cumulative execution time for the process.

COMMAND — The command and its arguments.

Ps makes an educated guess as to the file name and arguments given when the process was created by examining core memory or the swap area. The method is inherently somewhat unreliable and in any event a process is entitled to destroy this information, so the names cannot be counted on too much.

FILES

/unix	system namelist
/dev/mem	core memory

/sys/sys/core	alternate core file
/dev	searched to find swap device and tty names

SEE ALSO

kill(I)

NAME

ptx – permuted index

SYNOPSIS

ptx [**–t**] input [output]

DESCRIPTION

Ptx generates a permuted index from file *input* on file *output*. It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally the sorted lines are rotated so the keyword comes at the middle of the page.

Input should be edited to remove useless lines. The following words are suppressed: 'a', 'an', 'and', 'as', 'is', 'for', 'of', 'on', 'or', 'the', 'to', 'up'.

The optional argument **–t** causes *ptx* to prepare its output for the phototypesetter.

The index for this manual was generated using *ptx*.

FILES

/bin/sort

NAME

pump – Shell data transfer command

SYNOPSIS

pump [–[subchar]] [+] [eofstr]

DESCRIPTION

Pump is a filter that copies its standard input to standard output with possible substitution of Shell arguments and variables. It reads its input to end-of-file, or until it finds *eofstr* alone on a line. If not specified, *eofstr* is assumed to be '!'. Normally, Shell variable and argument values are substituted in the data stream, using '$' as the character to indicate their presence. The argument '–' alone suppresses all substitution, '–*subchar*' causes *subchar* to be used as the indicator character for substitution in place of '$'. Escaping is handled as in double quoted(") strings: the indicator character may be hidden by preceding it with a '\'. Otherwise, '\' and other characters are transmitted unchanged. The '+' flag causes all leading tab characters in the input to be thrown away, in order to permit readable indentation of text and *eofstr*. *Pump* may be used interactively and in pipelines. A common use is to get variable values into editor scripts. If $a, $b, and $c have the values A, B, and C respectively, the two sequences below are equivalent:

```
pump –~ | ed file          ed file
1,$s/~a$/~b/               1,$s/A$/B/
?~c?                       ?C?
!                          q
```

The sequence above will work at the terminal as well as in Shell procedures. *Pump* is an efficient and convenient replacement for multiple uses of *echo(I);* e.g., the following are equivalent:

```
pump >file                 echo "$1" >file
$1                         echo "$2" >>file
$2
!
```

Pump is actually implemented inside the Shell, although it executes as a separate process.

SEE ALSO

echo(I), sh(I)

BUGS

The size of *eofstr* is limited to 95 bytes, and it may not begin with '+'.

NAME

pwd — working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

SEE ALSO

chdir(I)

NAME

quiz – test your knowledge

SYNOPSIS

quiz [**–i** file] [**–t**] [category1 category2]

DESCRIPTION

Quiz gives associative knowledge tests on various subjects. It asks items chosen from *category1* and expects answers from *category2*. If no categories are specified, *quiz* gives instructions and lists the available categories.

Quiz tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, *quiz* reports a score and terminates.

The **–t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The **–i** flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

line	= category newline | category ':' line
category	= alternate | category '|' alternate
alternate	= empty | alternate primary
primary	= character | '[' category ']' | option
option	= '{' category '}'

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash '\' is used as with *sh*(I) to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, *quiz* will refrain from asking it.

FILES

/usr/lib/quiz/index
/usr/lib/quiz/*

NAME

rc – Ratfor compiler

SYNOPSIS

rc [–c] [–r] [–f] [–v] file ...

DESCRIPTION

Rc invokes the Ratfor preprocessor on a set of Ratfor source files. It accepts three types of arguments:

Arguments whose names end with '.r' are taken to be Ratfor source programs; they are preprocessed into Fortran and compiled. Each subroutine or function 'name' is placed on a separate file *name.f,* and its object code is left on *name.o.* The main routine is on *MAIN.f* and *MAIN.o;* block data subprograms go on *blockdata?.f* and *blockdata?.o.* The files resulting from a '.r' file are loaded into a single object file *file.o,* and the intermediate object and Fortran files are removed.

The following flags are interpreted by *rc.* See *ld*(I) for load-time flags.

–c Suppresses the loading phase of the compilation, as does any error in anything.

–f Save Fortran intermediate files. This is primarily for debugging.

–r Ratfor only; don't try to compile the Fortran. This implies **–f.**

–v Don't list intermediate file names while compiling.

Arguments whose names end with '.f' are taken to be Fortran source programs; they are compiled in the normal manner. (Only one Fortran routine is allowed in a '.f' file.)

Other arguments are taken to be either loader flag arguments, or Fortran-compatible object programs, typically produced by an earlier *rc* run, or perhaps libraries of Fortran-compatible routines. These programs, together with the results of any compilations specified, are loaded to produce an executable program with name **a.out.**

FILES

ratjunk	temporary
/usr/bin/ratfor	preprocessor
/usr/fort/fc1	Fortran compiler

SEE ALSO

RATFOR – A Preprocessor for a Rational Fortran by B. W. Kernighan.
fc(I) for Fortran error messages.

DIAGNOSTICS

Yes, both from *rc* itself and from Fortran.

BUGS

Limit of about 50 arguments, 10 block data files.

#define and #include lines in ".f" files are not processed.

NAME

reform – reformat text file

SYNOPSIS

reform [*tabspec1* [*tabspec2*]] [**+b***n*] [**+e***n*] [**+f**] [**+i***n*] [**+m***n*] [**+p***n*] [**+s**] [**+t***n*]

DESCRIPTION

Reform reads each line of the standard input file, reformats it, and then writes it to the standard output. Various combinations of reformatting operations can be selected, of which the most common involve rearrangement of tab characters. It is often used to trim trailing blanks, truncate lines to a specified length, or prepend blanks to lines.

Reform first scans its arguments, which may be given in any order. It then processes its input file, performing the following actions upon each line, in the order given:

- A line is read from the standard input.
- If **+s** is given, all characters up to the first tab are stripped off and saved for later addition to the end of the line. Presumably, these characters comprise an SCCS SID produced by *get*(I).
- The line is expanded into a tabless form, by replacing tabs with blanks according to the *input* tab specification *tabspec1*.
- If **+p***n* is given, *n* blanks are prepended to the line.
- If **+t***n* is given, the line is truncated to a length of *n* characters.
- All trailing blanks are now removed.
- If **+e***n* is included, the line is extended out with blanks to the length of *n* characters.
- If **+s** is given, the previously-saved SCCS SID is added to the end of the line.
- If **+b***n* is given, the *n* characters at the beginning of the line are converted to blanks, if and only if all of them are either digits or blanks.
- If **+m***n* is included, the line is moved left, i.e., *n* characters are removed from the beginning of the line.
- The line is now contracted by replacing some blanks with tab characters according to the list of tabs indicated by the *output* tab specification *tabspec2*, and is written to the standard output file. Option **+i** controls the method of contraction (see below).

The various arguments accepted by *reform* are as follows:

tabspec1 describes the tab stops assumed for the input file. This tab specification may take on any of the forms described in *tabs*(I). In addition, the operand "**--**" indicates that the tab specification is to be found in the first line read from the standard input. If no legal tab specification is found there, **-8** is assumed. If *tabspec1* is omitted entirely, "**--**" is assumed.

tabspec2 describes the tabs assumed for the output file. It is interpreted in the same way as *tabspec1*, except that omission of *tabspec2* causes the value of *tabspec1* to be used for *tabspec2*.

The remaining arguments are all optional and may be used in any combination, although only a few combinations make much sense. Specifying an argument causes an action to be performed, as opposed to the usual default of not performing the action. Some options include numeric values, which also have default values. Option actions are applied to each line in the order described

above. Any line length mentioned applies to the length of a line just before the execution of the option described, and the terminating newline is never counted in the line length.

+b*n* causes the first *n* characters of a line to be converted to blanks, if and only if those characters include only blanks and digits. If *n* is omitted, the default value is 6, which is useful in deleting sequence numbers from COBOL programs.

+e*n* causes each line shorter than *n* characters to be extended out with blanks to that length. Omitting *n* implies a default value of 72. This option is useful for those rare cases in which sequence numbers need to be added to an existing unnumbered file. The use of $ in editor regular expressions is more convenient if all lines have equal length, so that the user can issue editor commands such as:

s/$/00001000/

+f causes a format line to be written to the standard output, preceding any other lines written. See *fspec(V)* for details regarding format specifications. The format line is taken from *tabspec2*, i.e., the line normally appears as follows:

<:t−*tabspec2* **d:>**

If *tabspec2* is of the form **−−***filename* (i.e., an indirect reference to a tab specification in the first line of the named file), then that tab specification line is written to the standard output.

+i*n* controls the technique used to compress interior blanks into tabs. Unless this option is specified, any sequence of 1 or more blanks may be converted to a single tab character if that sequence occurs just before a tab stop. This causes no problems for blanks that occur before the first nonblank character in a line, and it is always possible to convert the result back to an equivalent tabless form. However, occasionally an interior blank (one occurring after the first nonblank) is converted to a tab when this is not intended. For instance, this might occur in any program written in a language utilizing blanks as delimiters. Any single blank might be converted to a tab if it occurred just before a tab stop. Insertion or deletion of characters preceding such a tab may cause it to be interpreted in an unexpected way at a later time. If the **+i** option is used, no string of blanks may be converted to a tab unless there are *n* or more contiguous blanks. The default value is 2. Note that leading blanks are always converted to tabs when possible. **It is recommended that conversion of programs from non-PWB to PWB systems use this option.**

+m*n* causes each line to be moved left *n* characters, with a default value of 6. This can be useful for crunching COBOL programs.

+p*n* causes *n* blanks to be prepended (default of 6 if *n* is omitted). This option is effectively the inverse of **+m***n*, and is often useful for adjusting the position of *nroff(I)* output for terminals lacking both forms tractor positioning and a settable left margin.

+s is used with the **−m** option of *get*(I). The **−m** option causes *get* to prepend to each generated line the appropriate SCCS SID, followed by a tab. The **+s** option causes *reform* to remove the SID from the front of the line, save it, then add it later to the end of the line. Because **+e72** is implied by this option, the effect is to produce 80-character card images with SCCS SID in columns 73–80. Up to 8 characters of the SID are shown; if it is longer, the eighth character is replaced by '*' and any characters to the right of it are discarded.

+t*n* causes any line longer than *n* characters to be truncated to that length. If *n* is omitted, the length defaults to 72. Sequence numbers can thus be removed and any blanks immediately preceding them deleted.

The following illustrate typical uses of *reform*. The terms "PWB" and "OBJECT" below refer to UNIX and non-UNIX computer systems, respectively. Each arrow indicates the direction of conversion. The character '?' indicates an arbitrary tab specification; see *tabs(I)* for descriptions of legal specifications.

OBJECT ----> PWB (i.e., manipulation of RJE output):

Note that files transferred by RJE from OBJECT to PWB materialize with format **-8**.

reform -8 ? +t +f <oldfile >newfile (into arbitrary format)
reform -8 -c +t +b +i <oldfile >newfile (into COBOL)
reform -8 -c3 +t +m +i <oldfile >newfile (into COBOL, crunched)

NOTE: -c3 is the preferred format for COBOL; it uses the least disk space of the COBOL formats.

PWB ----> OBJECT (i.e., preparation of files for RJE submission):

reform ? -8 <oldfile >newfile (from arbitrary format into -8)
get -p -m sccsfile | reform +s | send ...

PWB ONLY (i.e., no involvement with other systems):

pr file | reform ? -0 <oldfile (print on terminal without hardware tabs)
reform ? -0 <oldfile >newfile (convert file to tabless format)

DIAGNOSTICS

All diagnostics are fatal, and the offending line is displayed following the message.
"line too long" a line exceeds 512 characters (in tabless form).
"not SCCS -m" a line does not have at least one tab when +s flag is used.
Any of the diagnostics of *tabs(I)* can also appear.

EXIT CODES

0 - normal
1 - any error

SEE ALSO

fspec(V), get(I), nroff(I), send(I), tabs(I)

BUGS

Reform is aware of the meanings of backspaces and escape sequences, so that it can be used as a postprocessor for *nroff*. However, be warned that the **+e**, **+m**, **+t** options only count characters, not positions. Anyone using these options on output containing backspaces or halfline motions will probably obtain unexpected results.

NAME

regcmp – regular expression compile

SYNOPSIS

regcmp [–] file ...

DESCRIPTION

Regcmp, in most cases, precludes the need for calling *regcmp* (see *regex*(III)) from C programs. This saves on both execution time and program size. The command *regcmp* compiles the regular expressions in *file* and places the output in *file.i.* If the "–" option is used, the output will be placed in *file.c.*

The format of entries in *file* is a name (C variable), followed by one or more blanks, followed by a regular expression enclosed in double quotes. The output of *regcmp* is C source, which declares each variable name as an *extern char* array, and initializes that array with the compiled form of the corresponding regular expression. *File.i* files may thus be *included* into C programs, or *file.c* files may be compiled and later loaded. Diagnostics are self-explanatory.

Example:

```
name    "([A–Za–z][A–Za–z0–9_]*)$0"
telno   "\\({0,1}([2–9][01][1–9])$0\\){0,1} *"
        "([2–9][0–9]{2})$1[ –]{0,1}"
        "([0–9]{4})$2"
```

In the C program which uses the *regcmp* output,

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named *telno* to *line.*

SEE ALSO

regex(III)

NAME

rgrep – search a file for a pattern

SYNOPSIS

rgrep [**–v**] [**–b**] [**–c**] [**–n**] expression [file] ...

DESCRIPTION

Rgrep is an extended form of *grep* which uses the facilities of the *regex*(III) routine. *Rgrep* searches the input files (standard input default) for lines matching the regular expression. Normally, each line found is copied to the standard output. If the **–v** flag is used, all lines but those matching are printed. If the **–c** flag is used, only a count of matching lines is printed. If the **–n** flag is used, each line is preceded its relative line number in the file. If the **–b** flag is used, each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

In all cases the file name is shown if there is more than one input file.

For a complete description of the regular expression, see *ed*(I) and *regex*(III). Care should be taken when using the characters $ * [^ | ( ) and \ in the regular expression as they are also meaningful to the Shell. It is generally necessary to enclose the entire *expression* argument in quotes.

SEE ALSO

ed(I), sh(I), regex(III)

NAME

rjestat – RJE status and enquiries

SYNOPSIS

rjestat [–] [A] [B] [1110]

DESCRIPTION

When invoked without the '–' argument, *rjestat* reports the current status of RJE links to the specified host computers. When invoked with the '–' argument, *rjestat* sets up an interactive status terminal. If no hosts are cited explicitly, the specification defaults to all those for which a given PWB/UNIX is configured. The "host" pseudonyms **A**, **B**, and **1110** are built into the RJE software. **A** and **B** may be used to represent any IBM host machine. Their actual destinations are immaterial to RJE. The pseudonym **1110** is built into RJE to represent any UNIVAC host.

To enter an enquiry via such a status terminal, you must first generate an interrupt. This can be done by hitting the DEL key or the BREAK/INTERRUPT key. *Rjestat* will respond by prompting for enquiries directed to each host in turn. The line on which a prompt appears may be completed to form a legitimate display command for that particular host. If the line is terminated with a '\', the prompt will be repeated, otherwise it will advance to the next host. A carriage return alone indicates that no enquiry is to be directed to a particular host. You should expect to wait at least 30 seconds for a response.

An interrupt will temporarily halt the display of responses. It can therefore be used to inhibit roll-up on a CRT terminal. The display of responses will resume after all prompts have been satisfied (perhaps by null completions).

To exit from the status terminal, generate a quit signal or type DEL followed by EOT.

The UNIVAC 1110 capability is only supported at the BTL Piscataway location.

FILES

/dev/rje*	DQS-11's used by RJE
/usr/rje/sys	PWB/UNIX system name
/usr/rje/lines	configuration table

And, in the directory for each RJE subsystem:

log	activity log
resp	concatenated responses
status	message of the day
xmit*	files queued
*mesg	enquiry slot
*init	boot program

SEE ALSO

Guide to IBM Remote Job Entry for PWB/UNIX Users by A. L. Sabsevitz.
OS/VS2 HASP II Version 4 Operator's Guide, IBM SRL #GC27-6993.
Operator's Library: OS/VS2 Reference (JES2), IBM SRL #GC38-0210.

NAME

rm – remove (unlink) files

SYNOPSIS

rm [−f] [−r] name ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If the user does not have write permission on a file, *rm* prints the file name and its mode, then reads a line from the standard input. If the line begins with y, the file is removed, otherwise it is not. The question is not asked if option **−f** was given or if the standard input is not a terminal.

If a designated file is a directory, an error comment is printed unless the optional argument **−r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory. To remove directories *per se* see *rmdir*(I).

FILES

/etc/glob to implement the **−r** flag

SEE ALSO

rmdir(I)

BUGS

When *rm* removes the contents of a directory under the **−r** flag, full pathnames are not printed in diagnostics.

NAME

rmdel – remove a delta from an SCCS file

SYNOPSIS

rmdel –rSID name ...

DESCRIPTION

Rmdel removes the delta specified by the SID from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file.

If a directory is named, *rmdel* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with "s."), and unreadable files, are silently ignored. If a name of "–" is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files, and unreadable files, are silently ignored.

The exact permissions necessary to remove a delta are documented in the *SCCS/PWB User's Manual*. Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

x-file (see *delta*(I))
z-file (see *delta*(I))

SEE ALSO

get(I), delta(I), prt(I), help(I), sccsfile(V)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

rmdir – remove directory

SYNOPSIS

rmdir dir ...

DESCRIPTION

Rmdir removes (deletes) directories. The directory must be empty (except for the standard entries '.' and '..', which *rmdir* itself removes). Write permission is required in the directory in which the directory to be removed appears.

BUGS

Needs a **–r** flag.

Actually, write permission in the directory's parent is *not* required.

Mildly unpleasant consequences can follow removal of your own or someone else's current directory.

NAME

roff − format text

SYNOPSIS

roff [+*n*] [−*n*] [**−s**] [**−h**] file ...

DESCRIPTION

Roff formats text according to control lines embedded in the text in the given files. Encountering a nonexistent file terminates printing. Incoming interconsole messages are turned off during printing. The optional flag arguments mean:

+*n* Start printing at the first page with number *n*.

−*n* Stop printing at the first page numbered higher than *n*.

−s Stop before each page (including the first) to allow paper manipulation; resume on receipt of an interrupt signal.

−h Insert tabs in the output stream to replace spaces whenever appropriate.

Input consists of intermixed *text lines*, which contain information to be formatted, and *request lines*, which contain instructions about how to format it. Request lines begin with a distinguished *control character*, normally a period.

Output lines may be *filled* as nearly as possible with words without regard to input lineation. Line *breaks* may be caused at specified places by certain commands, or by the appearance of an empty input line or an input line beginning with a space.

The capabilities of *roff* are specified in the attached Request Summary. Numerical values are denoted there by n or +n, titles by t, and single characters by c. Numbers denoted +n may be signed + or −, in which case they signify relative changes to a quantity, otherwise they signify an absolute resetting. Missing n fields are ordinarily taken to be 1, missing t fields to be empty, and c fields to shut off the appropriate special interpretation.

Running titles usually appear at top and bottom of every page. They are set by requests like

.he 'part1'part2'part3'

Part1 is left justified, part2 is centered, and part3 is right justified on the page. Any % sign in a title is replaced by the current page number. Any nonblank may serve as a quote.

ASCII tab characters are replaced in the input by a *replacement character*, normally a space, according to the column settings given by a .ta command. (See .tr for how to convert this character on output.)

Automatic hyphenation of filled output is done under control of .hy. When a word contains a designated *hyphenation character*, that character disappears from the output and hyphens can be introduced into the word at the marked places only.

FILES

/usr/lib/suftab suffix hyphenation tables
/tmp/rtm? temporary

SEE ALSO

nroff(I), troff(I)

BUGS

Roff is the simplest of the run-off programs, but is utterly frozen and quite obsolescent.

REQUEST SUMMARY

Request	*Break*	*Initial*	*Meaning*
.ad	yes	yes	Begin adjusting right margins.
.ar	no	arabic	Arabic page numbers.
.br	yes	–	Causes a line break – the filling of the current line is stopped.
.bl n	yes	–	Insert of n blank lines, on new page if necessary.
.bp +n	yes	n=1	Begin new page and number it n; no n means '+1'.
.cc c	no	c=.	Control character becomes 'c'.
.ce n	yes	–	Center the next n input lines, without filling.
.de xx	no	–	Define parameterless macro to be invoked by request '.xx' (definition ends on line beginning '..').
.ds	yes	no	Double space; same as '.ls 2'.
.ef t	no	t=""	Even foot title becomes t.
.eh t	no	t=""	Even head title becomes t.
.fi	yes	yes	Begin filling output lines.
.fo	no	t=""	All foot titles are t.
.hc c	no	none	Hyphenation character becomes 'c'.
.he t	no	t=""	All head titles are t.
.hx	no	–	Title lines are suppressed.
.hy n	no	n=1	Hyphenation is done, if n=1; and is not done, if n=0.
.ig	no	–	Ignore input lines through a line beginning with '..'.
.in +n	yes	–	Indent n spaces from left margin.
.ix +n	no	–	Same as '.in' but without break.
.li n	no	–	Literal, treat next n lines as text.
.ll +n	no	n=65	Line length including indent is n characters.
.ls +n	yes	n=1	Line spacing set to n lines per output line.
.m1 n	no	n=2	Put n blank lines between the top of page and head title.
.m2 n	no	n=2	n blank lines put between head title and beginning of text on page.
.m3 n	no	n=1	n blank lines put between end of text and foot title.
.m4 n	no	n=3	n blank lines put between the foot title and the bottom of page.
.na	yes	no	Stop adjusting the right margin.
.ne n	no	–	Begin new page, if n output lines cannot fit on present page.
.nn +n	no	–	The next n output lines are not numbered.
.nl	no	no	Add 5 to page offset; number lines in margin from 1 on each page.
.n2 n	no	no	Add 5 to page offset; number lines from n; stop if n=0.
.ni +n	no	n=0	Line numbers are indented n.
.nf	yes	no	Stop filling output lines.
.nx filename		–	Change to input file 'filename'.
.of t	no	t=""	Odd foot title becomes t.
.oh t	no	t=""	Odd head title becomes t.
.pa +n	yes	n=1	Same as '.bp'.
.pl +n	no	n=66	Total paper length taken to be n lines.
.po +n	no	n=0	Page offset. All lines are preceded by n spaces.
.ro	no	arabic	Roman page numbers.
.sk n	no	–	Produce n blank pages starting next page.
.sp n	yes	–	Insert block of n blank lines, except at top of page.
.ss	yes	yes	Single space output lines, equivalent to '.ls 1'.
.ta n n..		–	Pseudotab settings. Initial tab settings are columns 9 17 25 ...
.tc c	no	space	Tab replacement character becomes 'c'.
.ti +n	yes	–	Temporarily indent next output line n spaces.
.tr cdef..	no	–	Translate c into d, e into f, etc.
.ul n	no	–	Underline the letters and numbers in the next n input lines.

NAME

rsh – restricted shell (command interpreter)

SYNOPSIS

rsh [−x] [−] [−ct] [name [arg1 ...]]

DESCRIPTION

Rsh is a restricted version of the standard command interpreter *sh(I)*. It is used to set up login names or execution environments whose capabilities are more controlled than that of the standard shell. The actions of *rsh* are identical to those of *sh*, except for the following restrictions:

1) *chdir* is not allowed.

2) changes to the shell variable '$p' are not permitted.

3) it is illegal to use '/' in the name of a command.

4) *next* is not permitted.

5) '>' and '>>' are disallowed.

These restrictions combine to lock a user into the login directory, limit the set of invokable commands to those found in directories included in the '.path' file, and eliminate the direct creation or modification of files. When a command to be executed is found to be a shell procedure, *rsh* invokes *sh* to execute it. Thus, it is possible to write shell procedures using the full power of the standard shell, while the end user is restricted to a limited menu of commands.

Rsh is actually just a link to *sh*.

FILES

/etc/glob, which interprets '*', '?', and '['.
/dev/null as a source of end-of-file.
.path in login directory to initialize $p.
.profile in login directory for general initialization.
/etc/sha for accounting information.

SEE ALSO

sh(I)

BUGS

It would be better to have a flag for *opt* which changed *sh* into *rsh* dynamically. With a non-interruptable '.profile', it would be possible to act as *sh*, use *chdir* (for example), and then change into *rsh* at the end of initialization.

NAME

sccsdiff – compare two versions of an SCCS file

SYNOPSIS

sccsdiff old-spec new-spec [pr-args] sccsfile ...

DESCRIPTION

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. The *old-spec* is any valid *get*(I) specifier (e.g., –r1.1) for the old version to be gotten. Similarly, *new-spec* is any valid *get*(I) specifier (e.g., –r1.4) for the new version to be gotten. The *pr-args* are any valid *pr*(I) arguments which begin with a "–", except for "–h" (the output of *sccsdiff* is piped through *pr*(I)). Any number of SCCS files may be specified, but the *old-spec* and *new-spec* apply to all files.

Sccsdiff is a simple shell procedure; interested persons should "cat /usr/bin/sccsdiff" to discover how it works.

FILES

/tmp/get?????	temporary for old gotten version
/usr/bin/bdiff	program that generates differences

SEE ALSO

get(I), help(I), pr(I), bdiff(I)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

DIAGNOSTICS

Use *help*(I) for explanations.

NAME

sed – stream editor

SYNOPSIS

sed [**−g**] [**−n**] [**−f** commandfile] ... [[**−e**] command] ... [file] ...

DESCRIPTION

Sed copies the input *file*s (default is standard input) to the standard output, perhaps performing one or more editor commands (see *ed*(I)) on each line.

The **−g** flag indicates that all *s* commands should be executed as though followed by a *g*. If only some substitutions are to be done globally, leave out the **−g** flag and put the *g*'s at the end of the appropriate command lines.

The **−n** flag indicates that only lines that are explicitly printed by *p* commands are to be copied to the standard output. In order to avoid getting double copies of some lines in the standard output, the *p* command is ignored unless the **−n** flag is set.

The **−e** flag indicates that the next argument is an editor command.

The **−f** flag indicates that the next argument is a file name; the file contains editor commands, one to a line. Commands that are inherently multi-line, like *a* or *c*, should have the interior newlines escaped by '\'. Append, insert, and change modes are terminated by an non-escaped newline.

The **−e** and **−f** flags may be intermixed in any order.

If no **−e** or **−f** flags are given, the first argument is taken by default to be an editor command.

Addresses are allowed. The meaning of *two* addresses is: "Attempt this command on the first line that matches the first address, and on all subsequent lines up to and including the first subsequent line that matches the second address; then search for a match of the first address and iterate." *One* address means: "Attempt this command on all lines that match the address." Either line-numbers or regular expressions are allowed as addresses. Line numbers increase monotonically throughout *all* the input files, so that, if *n* is the number of the last line of the first input file, then *n*+*1* is the number of the first line of the second file, etc. A '$' as an address matches the *last* line of the *last* input file.

The intention is to simulate the editor as exactly as possible, but the line-at-a-time operation makes certain differences unavoidable or desirable:

1. There is no notion of '.' and no relative addressing.
2. Commands with no addresses are defaulted to *1,$* rather than to dot.
3. Addresses specified as regular expressions must be delimited by '/'; '?' is an error.
4. Expressions in addresses are not allowed (i.e., '+', '−').
5. Commands may have only as many addresses as they can use. That is, no command may have more than two addresses; the *a*, *i*, and *r* commands may have only one address.
6. A *p* at the end of a command only works with the *s* command. For other commands, or if the **−n** flag is not in effect, a *p* at the end of a command line is ignored.
7. A *w* may appear at the end of a *s* command. It should be followed by a single space and a file name. If the *s* command succeeds, the modified line is appended to the file. All files are opened when the commands are being compiled, and closed when the program terminates. Only ten distinct file names may appear in *w* commands in a single execution of

sed. Unlike *p*, *w* takes effect regardless of the **-n** flag. If both *p* and *w* are appended to the same substitute command, they must be in the order *pw*.

8. The only editor commands available are *a*, *c*, *d*, *i*, *s*, *p*, *q*, *r*, *w*, *g*, *v*, and =. A successful execution of a *q* command causes the current line to be written out if it should be, and execution terminated. When a line is deleted by a *d* or *c* command, no further commands are attempted on its corpse, but another line is immediately read from the input (but see item 10. below).

9. The *n*ext line command, *n*, replaces the current line by the next line from the input file. The list of editing commands is continued after the *n* command is executed.

10. If an *a*, *i*, or *r* command is successfully executed, the text is inserted into the standard output whether or not the line on which the match was made is later deleted or not. Thus the commands:

```
/b/a\
XXX
/b/,/c/d
```

applied to the file

```
a
b
c
d
```

will produce

```
a
XXX
d
```

on the output.

11. Text inserted in the output stream by the *a*, *i*, *c*, or *r* commands is not scanned for any pattern matches, nor are any editor commands applied to it.

Sed supports three commands to control the flow of processing. These commands do no editing on the input line, but serve to control the order in which multiple editing commands are applied to an input line.

12. The label command, *: label*, marks a place in the list of editing commands which may be referred to by *j* and *t* commands (see 13. and 14. below); the *label* may be any sequence of eight or fewer characters; if two different colon commands have identical labels, a compile-time diagnostic will be generated and no execution attempted.

13. The *j*ump command, *j label*, causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon command with the same *label* was encountered. If no colon command with the same label can be found after all editing commands have been compiled, a compile-time diagnostic is produced and no execution is attempted. A *j* command with no *label* is taken to be a jump to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning of that line.

14. The *t*est command, *t label*, tests whether *any* successful substitutions have been made on the current input line; if so, it jumps to *label*; if not, it does nothing. The flag that indicates that a successful substitution has occurred on the current input line is reset by either reading a new line or by executing the *t* command.

Sed also supports command grouping and several operations that can build lines into a pattern space to be operated upon by other commands.

15. Commands may be grouped by curly braces. The opening brace must appear in the place where a command would ordinarily appear; the closing brace must appear on a line by itself (except for leading blanks or tabs). If the first line of a command file has *#n* as its first two characters, the no-copy flag is set, as though the **−n** option had been given on the command line. The remainder of this first line is ignored and may be used for a title or a comment. As an example:

```
#n  Print first non-blank line after a blank line, and first line, if non-blank.
1{
        /./p
}
/^$/{
: loop
        n
        /./{
                p
                j
        }
        j loop
}
```

16. The *N*ext command, *N*, appends the next input line to the current line; the two lines are separated by a new-line character, that may be matched by '\n'.

17. The *D*elete command, *D*, deletes up to and including the first (leftmost) new-line in the current pattern space. If the pattern space becomes empty (the only new-line is at the end of the space), *D*elete reads another line from the input. The list of editing commands is restarted from the beginning.

18. The *P*rint command, *P*, prints on standard output up to and including the first new-line in the pattern space.

SEE ALSO

ed(I)

BUGS

Lines are silently truncated to a maximum length of 512 characters. The "plus", "range", and "through" regular expression operators ("+", "\{\}", "[−]") of *ed*(I) are not implemented in *sed*.

NAME

send – submit RJE job

SYNOPSIS

send argument ...

DESCRIPTION

Send is a command-level interface to the RJE subsystems *hasp*(VIII) and *uvac*(VIII). It allows the user to collect input from various sources in order to create a run stream consisting of card images. *Send* creates a temporary file, with a special format, to contain the collected run stream, and then queues the file for transmission by invoking *haspqer* or *uvacqer*, as appropriate. Further processing of the job is controlled by the appropriate PWB/UNIX RJE subsystem and the host computer to which the job is submitted.

Possible sources of input to *send* are: ordinary files, standard input, the terminal, and the output of a command or shell file. Each source of input is treated as a virtual file, and no distinction is made based upon its origin. Typical input is an ASCII text file of the sort that is created by the editor *ed*(I). An optional format specification appearing in the first line of a file (see */spec*(V)) determines the settings according to which tabs are expanded into spaces. In addition, lines that begin with "~" are normally interpreted as commands controlling the execution of *send*. They may be used to set or reset flags, to define keyword substitutions, and to open new sources of input in the midst of the current source. Other text lines are translated one-for-one into card images of the run stream.

The run stream that results from this collection is treated as one job by the RJE subsystems. *Send* provides a card count for the run stream, and the queuer that is invoked announces the position that the job has been assigned in the queue of jobs waiting to be transmitted. The initial card of a job submitted to an IBM system must have a "/" in the first column. The initial card of a job submitted to a UNIVAC system must begin with a "@RUN" or "'run", etc. Any cards preceding these will be excised. If a host computer is not specified before the first card of the runstream is ready to be sent, *send* will select a reasonable default. In the case of an IBM job, all cards beginning "/*$" will be excised from the runstream, because they are HASP command cards.

The arguments that *send* accepts are described below. An argument is interpreted according to the first pattern that it matches. Preceding a character with "\" causes it to lose any special meaning it might otherwise have when matching against an argument pattern.

.	Close the current source.
–	Open standard input as a new source.
+	Open the terminal as a new source.
:*spec*:	Establish a default format specification for included sources, e.g., :m6t-12:.
:*message*	Print message on the terminal.
–:*prompt*	Open standard input and, if it is a terminal, print *prompt*.
+:*prompt*	Open the terminal and print *prompt*.
–*flags*	Set the specified flags, which are described below.
+*flags*	Reset the specified flags.

=flags	Restore the specified flags to their state at the previous level.
!command	Execute the specified PWB/UNIX *command* via the one-line Shell, with input redirected to */dev/null* as a default. Open the standard output of the command as a new source.
$line	Collect contiguous arguments of this form and write them as consecutive lines to a temporary file; then have the file executed by the Shell. Open the standard output of the Shell as a new source.
"comment	Ignore this argument.
=:keyword	Prompt for a definition of *keyword* from the terminal.
keyword=^xx	Define *keyword* as a two-digit hexadecimal character code.
keyword=string	Define *keyword* in terms of a replacement string.
host	Job is to be submitted to: **A**, **B**, **1110**. The pseudonyms **A** and **B** are built into RJE to represent any IBM host connection. Their actual destinations are immaterial to RJE. The pseudonym **1110** is built into RJE to represent any UNIVAC host.
filename	Open the specified file as a new source of input.

Arguments of the form "*!chdir directory*" will be trapped so that the *send* process can execute the specified *chdir* itself. The original directory will be restored at the end of any source that contains a *chdir*.

The flags recognized by *send* are described in terms of the special processing that occurs when they are set:

- **−l** List card images on standard output. EBCDIC characters are translated back to ASCII.
- **−q** Do not output card images.
- **−f** Do not fold lower case to upper.
- **−t** Trace progress on diagnostic output, by announcing the opening of input sources.
- **−k** Ignore the keywords that are active at the previous level and erase any keyword definitions that have been made at the current level.
- **−r** Process included sources in raw mode; pack arbitrary 8-bit bytes one per column (80 columns per card) until an end-of-file.
- **−i** Do not interpret control lines in included sources; treat them as text.
- **−s** Make keyword substitutions before detecting and interpreting control lines.
- **−y** Suppress error diagnostics and submit job anyway.
- **−g** Gather mode, qualifying **−l** flag; list text lines before converting them to card images.
- **−h** Write listing with standard tabs.
- **−p** Prompt with "*" when taking input from the terminal.
- **−m** When input returns to the terminal from a lower level, repeat the prompt, if any.
- **−a** Make **−k** flag propagate to included sources, thereby protecting them from keyword substitutions.

−c List control lines on diagnostic output.

−d Extend the current set of keyword definitions by adding those active at the end of included sources.

Control lines are input lines that begin with "~". In the default mode **+ir**, they are interpreted as commands to *send*. Normally they are detected immediately and read literally. The **−s** flag forces keyword substitutions to be made before control lines are intercepted and interpreted. Arguments appearing in control lines are handled exactly like the command arguments to *send*, except that they are processed at a nested level of input.

The two possible formats for a control line are: "~argument" and "~ argument ...". In the first case, where the "~" is not followed by a space, the remainder of the line is taken as a single argument to *send*. In the second case, the line is parsed to obtain a sequence of arguments delimited by spaces. In this case the quotes "'" and "\"" may be employed to pass embedded spaces.

The interpretation of the argument "." is chosen so that an input line consisting of "~." is treated as a logical end-of-file. The following example illustrates some of the above conventions:

```
send −
~ argument ...
~.
```

This sequence of three lines is equivalent to the command synopsis at the beginning of this description. In fact, the "−" is not even required. By convention, the *send* command reads standard input if no other input source is specified. *Send* may therefore be employed as a filter with side-effects.

The execution of the *send* command is controlled at each instant by a current environment, which includes the format specification for the input source, a default format specification for included sources, the settings of the mode flags, and the active set of keyword definitions. This environment can be altered dynamically. When a control line opens a new source of input, the current environment is pushed onto a stack, to be restored when input resumes from the old source. The initial format specification for the new source is taken from the first line of the file. If none is provided, the established default is used or, in its absence, standard tabs. The initial mode settings and active keywords are copied from the old environment. Changes made while processing the new source will not affect the environment of the old source, with one exception: if **−d** mode is set in the old environment, the old keyword context will be augmented by those definitions that are active at the end of the new source. When *send* first begins execution, all mode flags are reset, and no keywords are defined.

The initial, reset state for all mode flags is the "+" state. In general, special processing associated with a mode *x* is invoked by flag *−x* and is revoked by flag *+x*. Most mode settings have an immediate effect on the processing of the current source. Exceptions to this are the **−r** and **−i** flags, which apply only to included source, causing it to be processed in an uninterpreted manner.

A keyword is an arbitrary ASCII string for which a replacement has been defined. The replacement may be another string, or (for IBM RJE only) the hexadecimal code for a single 8-bit byte. At any instant, a given set of keyword definitions is active. Input text lines are scanned, in one pass from left to right, and longest matches are attempted between substrings of the line and the active set of keywords. Characters that do not match are output, subject to folding and the standard translation. Keywords are replaced by the specified hexadecimal code or replacement string, which is then output character by character. The expansion of tabs and length checking, according to the format specification of an input source, are delayed until substitutions have been made in a line.

All of the keywords definitions made in the current source may be deleted by setting the **−k** flag. It then becomes possible to reuse them, although this is not recommended. Setting the **−k** flag also causes keyword definitions active at the previous source level to be ignored. Setting the **+k** flag causes keywords at the previous level to be ignored but does not delete the definitions made at the current level. The **=k** argument reactivates the definitions of the previous level.

A keyword may not be redefined, except redundantly, if it is active at some level of source input and its replacement is not null. Prompts for keywords that have already been defined at some higher level will simply cause the definitions to be copied down to the current level; new definitions will not be solicited. Only in the case where a keyword is defined by a null replacement, *A*=, is a redefinition allowed, *A*=*a*. Prompts for the keyword, =:*A*, will be satisfied by either definition.

Keyword substitution is an elementary macro facility that is easily explained and that appears useful enough to warrant its inclusion in the *send* command. More complex replacements are the function of a general macro processor(*m4*(I), perhaps. To reduce the overhead of string comparison, it is recommended that keywords be chosen so that their initial characters are unusual. For example, let them all be upper case.

Send performs two types of error checking on input text lines. Firstly, only ASCII graphics and tabs are permitted in input text. Secondly, the length of a text line, after substitutions have been made, may not exceed 80 bytes for IBM, or 132 bytes for UNIVAC. The length of each line may be additionally constrained by a size parameter in the format specification for an input source. Diagnostic output provides the location of each erroneous line, by line number and input source, a description of the error, and the card image that results. Other routine errors that are announced are the inability to open or write files, and abnormal exits from the Shell. Normally, the occurrence of any error causes *send,* before invoking the queuer, to prompt for positive affirmation that the suspect run stream should be submitted.

The *hasp* subsystem, which supports IBM RJE, operates in EBCDIC code. The *send* command is therefore required to translate ASCII characters into their EBCDIC equivalents. The standard conversion is based on the character set described in "Appendix H" of *IBM System/370 Principles of Operation* (IBM SRL GA22-7000). Each ASCII character in the octal range 040-176 possesses an EBCDIC graphic equivalent into which it is mapped, with four exceptions: broken vertical bar into "|", "^" into "¬", "[" into "¢", "]" into broken vertical bar. In listings requested from *send* and in printed output returned by *hasp,* the reverse translation is made from EBCDIC to ASCII, with the qualification that EBCDIC codes that do not have ASCII equivalents are translated into "^". The *uvac* subsystem, on the other hand, operates in ASCII code, and any translations between ASCII and field-data are made, in accordance with the UNIVAC standard, by the host computer.

Additional control over the translation process is afforded by the **−f** flag and hexadecimal character codes. As a default, *send* folds lower-case letters into upper case. For UNIVAC RJE it does more: the entire ASCII range 140-176 is folded into 100-136, so that "`", for example, becomes "@". In either case, setting the **−f** flag inhibits any folding. Non-standard character codes are obtained as a special case of keyword substitution.

When invoked under the name *gath,* the *send* command establishes initial flag settings **−lgq** and suppresses announcement of a zero card count. While in **−gq** mode, long lines that are detected elicit a diagnostic but are not truncated. Also, in this mode, it is potentially useful to convey non-graphics to standard output. To prevent *gath* from deleting non-printing characters, each may be declared as a single character keyword whose replacement is itself. To retain backspaces, for example, supply the argument "BS=BS", where BS denotes the ASCII character whose octal code is 010.

The UNIVAC 1110 capability is only supported at the BTL Piscataway location.

FILES

/bin/sh	Shell
/tmp/sh*	Shell temporary
/usr/rje/sys	PWB/UNIX system name, e.g., "A"
/usr/rje/lines	RJE configuration table

And, where *xxxx* is either *hasp* or *uvac:*

/usr/xxxx/pool/stm*	temporary
/usr/xxxx/xmit???	queued output
/usr/xxxx/xxxxqer	queueing program
/usr/xxxx/xxxxlock	null file for lockout
/usr/xxxx/xxxxstat	queue status record

SEE ALSO

help(1), m4(1), sh(1), ascii(V), ebcdic(V), fspec(V), hasp(VIII)
Guide to IBM Remote Job Entry for PWB/UNIX Users by A. L. Sabsevitz.

DIAGNOSTICS

"non-graphic deleted", "undefined tab deleted", "long line detected", "long line truncated", "illegal card excised" – followed by the resulting card image.
"Errors detected" – type "y" to submit anyway.

Use *help*(1) for explanations of error messages.

BUGS

Standard input is read in blocks, and unused bytes are returned via *seek*(II). If standard input is a pipe, multiple arguments of the form "–" and "–:*prompt*" should not be used, nor should the logical end-of-file "^.".

NAME

sh – shell (command interpreter)

SYNOPSIS

sh [−v] [−] [−ct] [name [arg1 ...]]

DESCRIPTION

Sh is the standard command interpreter. It is the program which reads and arranges the execution of the command lines typed by most users. It may itself be called as a command to interpret files of commands. Before discussing the arguments to the Shell when it is used as a command, the structure of command lines themselves will be given.

Commands. Each command is a sequence of non-blank command arguments separated by blanks. The first argument specifies the name of a command to be executed. Except for certain types of special arguments discussed below, the arguments other than the command name are passed without interpretation to the invoked command.

By default, if the first argument is the name of an executable file, it is invoked; otherwise the string "/bin/" is prepended to the argument. (In this way most standard commands, which reside in "/bin", are found.) If no such command is found, the string "/usr" is further prepended (to give "/usr/bin/command") and another attempt is made to execute the resulting file. (Certain lesser-used commands live in "/usr/bin".) If a command name contains a "/", it is invoked as is, and no prepending ever occurs. This standard command search sequence may be changed by the user. See the description of the Shell variable "$p" below.

If a non-directory file exists that matches the command name and has executable mode, but not the form of an executable program (does not begin with the proper magic number) then it is assumed to be an ASCII file of commands and a new Shell is created to execute it. See "Argument passing" below.

If the file cannot be found, a diagnostic is printed.

Command lines. One or more commands separated by "|" or "^" constitute a chain of *filters*, or a *pipeline*. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected by pipes (see *pipe*(II)) to its neighbors. A command line contained in parentheses "( )" may appear in place of a simple command as a filter.

A *command line* consists of one or more pipelines separated, and perhaps terminated by ";" or "&", or separated by "||" or "&&". The semicolon designates sequential execution. The ampersand causes the following pipeline to be executed without waiting for the preceding pipeline to finish. The process id of the preceding pipeline is reported, so that it may be used if necessary for a subsequent *wait* or *kill*. A pipeline following "&&" is executed only if the preceding pipeline completed successfully (exit code zero), while that following "||" is executed only if the preceding one did *not* execute successfully (exit code non-zero). The exit code tested is that of the last command in the pipeline. The "&&" operator has higher precedence.

Termination Reporting. If a command (not followed by "&") terminates abnormally, a message is printed. (All terminations other than exit and interrupt are considered abnormal). Termination reports for commands followed by "&" are given upon receipt of the first command subsequent to the termination of the command, or when a *wait* is executed. The following is a list of the abnormal termination messages:

Bus error
Trace/BPT trap

Illegal instruction
IOT trap
EMT trap
Bad system call
Quit
Floating exception
Memory violation
Killed
Broken Pipe
Alarm clock
Terminated

If a core image is produced, "– Core dumped" is appended to the appropriate message.

Redirection of I/O. There are three character sequences that cause the immediately following string to be interpreted as a special argument to the Shell itself. Such an argument may appear anywhere among the arguments of a simple command, or before or after a parenthesized command list, and is associated with that command or command list.

An argument of the form "<arg" causes the file "arg" to be used as the standard input (file descriptor 0) of the associated command.

An argument of the form ">arg" causes file "arg" to be used as the standard output (file descriptor 1) for the associated command. "Arg" is created if it did not exist, and in any case is truncated at the outset.

An argument of the form ">>arg" causes file "arg" to be used as the standard output for the associated command. If "arg" did not exist, it is created; if it did exist, the command output is appended to the file.

For example, either of the command lines

```
ls >junk; cat tail >>junk
( ls; cat tail ) >junk
```

creates, on file "junk", a listing of the working directory, followed immediately by the contents of file "tail".

Either of the constructs ">arg" or ">>arg" associated with any but the last command of a pipeline is ineffectual, as is "<arg" in any but the first.

In commands called by the Shell, file descriptor 2 refers to the standard output of the Shell regardless of any redirection of standard output. Thus filters may write diagnostics to a location where they have a chance to be seen.

A redirection of the form "<——" requests input from the standard input that existed when the instance of the Shell was created. This permits a command file to be treated as a filter. The procedure "lower" could be used in a pipeline to convert characters to lower case:

```
tr "[A–Z]" "[a–z]" <——
```

A typical invocation might be:

```
reform –8 –c <prnt0 | lower >prnt0a
```

Generation of argument lists. If any argument contains any of the characters "?", "*" or "[", it is treated specially as follows. The current directory is searched for files which *match* the given argument.

The character "*" in an argument matches any string of characters in a file name (including the null string).

The character "?" matches any single non-null character in a file name.

Square brackets "[...]" specify a class of characters which matches any single file name character in the class. Within the brackets, each ordinary character is taken to be a member of the class. A pair of characters separated by "-" places in the class each character lexically greater than or equal to the first and less than or equal to the second member of the pair.

Other characters match only the same character in the file name.

If an argument starts with "*", "?", or "[", that argument will not match any file name that starts with ".".

For example, "*" matches all file names; "?" matches all one-character file names; "[ab]*.s" matches all file names beginning with "a" or "b" and ending with ".s"; "?[zi-m]" matches all two-character file names ending with "z" or the letters "i" through "m". None of these examples match names that start with ".".

If the argument with "*" or "?" also contains a "/", a slightly different procedure is used: instead of the current directory, the directory used is the one obtained by taking the unmodified argument to the "/" preceding the first "*?[". The matching process matches the remainder of the argument after this "/" against the files in the derived directory. For example: "/usr/dmr/a*.s" matches all files in directory "/usr/dmr" which begin with "a" and end with ".s".

In any event, a list of names is obtained which match the argument. This list is sorted into alphabetical order, and the resulting sequence of arguments replaces the single argument containing the "*", "[", or "?". The same process is carried out for each argument (the resulting lists are *not* merged) and finally the command is called with the resulting list of arguments.

If a command has one argument with "*", "?", or "[", a diagnostic is printed if no file names match that argument. If a command has several such arguments, a diagnostic is only printed if they *all* fail to match any files.

Quoting. The character "\" causes the immediately following character to lose any special meaning it may have to the Shell; in this way "<", ">", and other characters meaningful to the Shell may be passed as part of arguments. A special case of this feature allows the continuation of commands onto more than one line: a new-line preceded by "\" is translated into a blank.

A sequence of characters enclosed in single quotes (') is taken literally, with no substitution or special processing whatsoever.

Sequences of characters enclosed in double quotes (") are also taken literally, except that "\", "", and "$" are handled specially. The sequences "\"" and "\$" yield "" and "$", respectively. The sequence "\x", where "x" is any character except "" or "$", yields "\x". A "$" within a quoted string is processed in the same manner as a "$" that is not in a quoted string (see below), unless it is preceded by a "\". For example:

```
ls | pr -h "\\"My directory\"\$"
```

causes a directory listing to be produced by *ls*, and passed on to *pr* to be printed with the heading "\"My directory"$". Quotes permit the inclusion of blanks in the heading, which is a single argument to *pr*. Note that "\" inside quotes disappears only when preceding "$" or "".

Argument passing. When the Shell is invoked as a command, it has additional string processing capabilities. Recall that the form in which the Shell is invoked is

sh [−v] [name [arg1 ...]]

The *name* is the name of a file which is read and interpreted. If not given, this subinstance of the Shell continues to read the standard input file.

In command lines in the file (and also in command input), character sequences of the form "$N", where *N* is a digit, are replaced by the *n*th argument to the invocation of the Shell (*argn*). "$0" is replaced by *name*. Shell variables ("$a" − "$z"), described below, are replaced in the same way.

The special argument "$*" is a name for the *current* sequence of all arguments from "$1" through the last argument, each argument separated from the previous by a single blank.

The special argument "$$" is the ASCII representation of the unique process number of the current Shell. This string is useful for creating temporary file names within command files.

The sequence "$x", where "x" is any character except one of the 38 characters mentioned above, is taken to refer to a variable "x" whose value is the null string. All substitution on a command line occurs *before* the line is interpreted: no action that alters the value of any variable can have any effect on a reference to that variable that occurs on the *same* line.

The argument **−t**, used alone, causes *sh* to read the standard input for a single line, execute it as a command, and then exit. It is useful for interactive programs which allow users to execute system commands.

The argument **−c** (used with one following argument) causes the next argument to be taken as a command line and executed. No new-line need be present, but new-line characters are treated appropriately. This facility is useful as an alternative to **−t** where the caller has already read some of the characters of the command to be executed.

The argument **−v** ("verbose") causes every command line to be printed after all substitution occurs, but before execution. Each argument is preceded by a single blank. When given, the **−v** must be the first argument.

Used alone, the argument "−" suppresses prompting, and is commonly used when piping commands into the Shell:

ls | sed "s/.*/echo &:;cat &/" | sh −

prints all files in a directory, each prefaced by its name.

Initialization. When the Shell is invoked under the name "−" (as it is when you login), it attempts to read the file ".profile" in the current directory and execute the commands found there. When it finishes with ".profile", the Shell prompts the user for input as usual. Typical files contain commands to set terminal tabs and modes, initialize values of Shell variables, look at mail, etc.

End of file. An end-of-file in the Shell's input causes it to exit. A side effect of this fact means that the way to log out from UNIX is to type an EOT.

Command file errors; interrupts. Any Shell-detected error, or an interrupt signal, during the execution of a command file causes the Shell to cease execution of that file. (Except after *onintr;* see below.)

Processes that are created with "&" ignore interrupts. Also if such a process has not redirected its input with a "<", its input is automatically redirected to come from the zero length file "/dev/null".

Special commands. The following commands are treated specially by the Shell. These commands generally do not work when named as arguments to programs like *time, if,* or *nohup*

because in these cases they are not invoked directly by the Shell.

chdir and *cd* are done without spawning a new process by executing *chdir*(II).

login is done by executing "/bin/login" without creating a new process.

wait is done without spawning a new process by executing *wait*(II).

shift [*integer*] is done by manipulating the arguments to the Shell. In the normal case, *shift* has the effect of decrementing the Shell argument names by one ("$1" disappears, "$2" becomes "$1", etc.). When the optional *integer* is given, only arguments equal to or greater than that number are shifted.

":" is simply ignored.

"=" *name* [*arg1* [*arg2*]]
The single character Shell variable *(name)* is assigned a value, either from the optional argument(s), or from standard input. If a single argument is given, its value is used. If a second argument is included, its value is used only if the first argument has a null value. This permits a simple way of setting up default values for arguments:

```
= a "$1" default
```

causing default to be used if "$1" is null or omitted entirely.

Such variables are referred to later with a "$" prefix. The variables "$a" through "$m" are guaranteed to be initialized to null, and will never have special meanings. The variables "$n" through "$z" are *not* guaranteed to be initialized to null, and may, at some time in the future, acquire special meanings. Currently, these variables have predefined meanings:

$n is the argument count to the Shell command.

$p contains the Shell directory search sequence for command execution. Alternatives are separated by ":". The default initial value is:

```
= p ":/bin:/usr/bin"
```

which executes from the current directory (the null pathname), then from "/bin", then from "/usr/bin", as described above. For the super-user, the value is:

```
= p "/bin:/etc:/"
```

Using the same syntax, users may choose their own sequence by storing it in a file named ".path" in their login directory. The ".path" information is available to successive Shells; the "$p" value is not. If the ".path" file contains a second line, it is interpreted as the name of the Shell to be invoked to interpret Shell procedures. (See "$z" below).

$r is the exit status code of the preceding command. "0" is the normal return from most commands.

$s is your login directory.

$t is your login tty letter.

$w is your file system name (first component of "$s").

$z is the name of the program to be invoked when a Shell procedure is to be executed. Its default value is "/bin/sh", but it can be overridden by supplying a second line in the ".path" file. It can be used to achieve consistent use of a specific Shell during periods when several distinct Shells are present in the system. For safety in the presence of change, use "$z" as a command rather than "sh".

No substitution of variables (or arguments) occurs within single quotes ('). Within double quotes ("), a variable string is substituted unchanged, even if it contains characters ("'", "\", or "$") that might otherwise be treated specially. In particular, the argument "$1" can be passed

unchanged to another command by using ""$1"". Outside quotes, substituted characters possess the same special meanings they have as if typed directly.

To illustrate, suppose that the shell procedure "mine" is called with two arguments:

```
sh  mine  'a; echo "$2"'  '"'
```

Then sample commands in "mine" and their output are as follows:

```
echo '$1'                   $1
echo "$1"                   a; echo "$2"
echo $1                     a
                            $2
echo $2a"                   a
echo "$2a"                  "a
echo $2                     syntax error
```

The appearance of the string "$2" (rather than "") occurs because the Shell performs only one level of substitution, i.e., no rescanning is done.

onintr [*label*]
Causes control to pass to the label named (using a *goto* command) if the Shell command file is interrupted. After such a transfer, interrupts are re-enabled. *Onintr* without an argument also enables interrupts. The special label "−" will cause any number of interrupts to be ignored.

next [*name*]
This command causes *name* to become the standard input. Current input is never effectively resumed. If the argument is omitted, your terminal keyboard is assumed.

pump [−[*subchar*]] [+] [*eofstr*]
This command reads its standard input until it finds *eofstr* (defaults to "!" if not specified) alone on a line. It normally substitutes the values of arguments and variables (marked with "$" as usual). If "−" is given alone, substitution is suppressed, and "−*subchar*" causes *subchar* to be used in place of "$" as the indicator character for substitution. Escaping is handled as in quoted strings: the indicator character may be escaped by preceding it by "\". Otherwise, "\" and other characters are transmitted unchanged. If "+" is used, leading tabs in the input are thrown away, allowing indentation. This command may be used interactively and in pipelines.

opt [−v] [+v] [−p *prompt-str*]
The argument −v turns on tracing, in the same style as a −v argument for the Shell. The argument +v turns it off. The argument −p causes the next argument string to be used as the prompt string for an interactive shell.

Commands implementing control structure. Control structure is provided by a set of commands that happen currently to be built into the Shell, although no guarantee is given that this will remain so. They are documented separately as follows:

if(I) − if, else, endif, and test.
switch(I) − switch, breaksw, endsw.
while(I) − while, end, break, continue.
goto(I) − goto.
exit(I) − exit.

FILES

/etc/sha, for shell accounting.
/dev/null as a source of end-of-file.
.path in login directory to initialize $p and name of Shell.
.profile in login directory for general initialization.

SEE ALSO

The UNIX Time-Sharing System by D. M. Ritchie and K. Thompson, CACM, July, 1974, which gives the theory of operation of the Shell.
PWB/UNIX Shell Tutorial by J. R. Mashey.
chdir(I), equals(I), exit(I), expr(I), fd2(I), if(I), login(I), loginfo(I), onintr(I), pump(I), shift(I), switch(I), wait(I), while(I), pexec(III), sha(V), glob(VIII)

EXIT CODE

If an error occurs in a command file, the Shell returns the exit value "1" to the parent process. Otherwise, the current value of the Shell variable *$r* is returned. Execution of a command file is terminated by an error.

BUGS

There is no built-in way to redirect the diagnostic output; *fd2(I)* must be used.
A single command line is limited to 1000 total characters, 50 arguments, and approximately 20 operators.

NAME

shift – adjust Shell arguments

SYNOPSIS

shift [digit]

DESCRIPTION

Shift is used in Shell command files to shift the argument list left by 1, so that old **$2** can now be referred to by **$1** and so forth. *Shift* is useful to iterate over several arguments to a command file. For example, the command file

```
while "$1"
        pr -3 $1
        shift
end
```

prints each of its arguments in 3-column format.

Shift is executed within the Shell.

The optional argument causes *shift* to leave shell arguments numbered lower than *$digit* alone on shifts; *shift* alone and *shift 1* are identical in effect.

SEE ALSO

sh(I)

NAME

size – size of an object file

20864

SYNOPSIS

size [object ...]

DESCRIPTION

Size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in octal and decimal, of each object-file argument. If no file is specified, **a.out** is used.

SEE ALSO

a.out(V)

NAME

sleep — suspend execution for an interval

SYNOPSIS

sleep time

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command in a certain amount of time as in:

```
(sleep 105; command)&
```

Or to execute a command every so often, as in this shell command file:

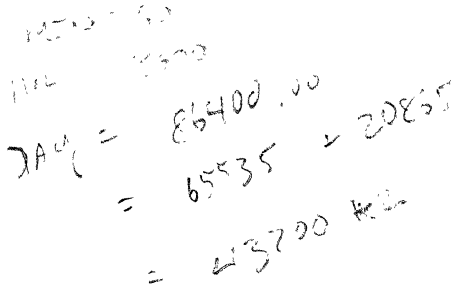
```
while 1
        command
        sleep 37
end
```

SEE ALSO

sleep(I)

BUGS

Time must be less than 65536 seconds.



NAME

sno – Snobol interpreter

SYNOPSIS

sno [file]

DESCRIPTION

Sno is a Snobol III (with slight differences) compiler and interpreter. *Sno* obtains input from the concatenation of *file* and the standard input. All input through a statement containing the label 'end' is considered program and is compiled. The rest is available to 'syspit'.

Sno differs from Snobol III in the following ways.

There are no unanchored searches. To get the same effect:

```
a ** b          unanchored search for b
a *x* b = x c   unanchored assignment
```

There is no back referencing.

```
x = "abc"
a *x* x         is an unanchored search for 'abc'
```

Function declaration is different. The function declaration is done at compile time by the use of the label 'define'. Thus there is no ability to define functions at run time and the use of the name 'define' is preempted. There is also no provision for automatic variables other than the parameters. For example:

define f( )

or

define f(a,b,c)

All labels except 'define' (even 'end') must have a non-empty statement.

If 'start' is a label in the program, program execution will start there. If not, execution begins with the first executable statement. 'define' is not an executable statement.

There are no builtin functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators '/' and '*' must be set off by space.

The right side of assignments must be non-empty.

Either '' or "" may be used for literal quotes.

The pseudo-variable 'sysppt' is not available.

SEE ALSO

Snobol III Manual (JACM Vol. 11, No. 1; Jan. 1964; pp. 21ff.)

NAME

sort – sort or merge files

SYNOPSIS

sort [**−mubdfinr**] [**−tx**] [**+**pos [**−**pos]] ... [**−o** name] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '−' means the standard input. The standard input is also used if no input file names are given. Thus *sort* may be used as a filter.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected by the following flags one or more of which may appear.

b Leading blanks (spaces and tabs) are not included in keys.
d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.
f Fold lower case letters onto upper case.
i Ignore all nonprinting nonblank characters in nonnumeric comparisons.
n An initial numeric string, consisting of optional minus sign, digits and optionally included decimal point, is sorted by arithmetic value.
r Reverse the sense of comparisons.
t*x* Tab character between fields is *x*.

Selected parts of the line, specified by **+***pos* and **−***pos*, may be used as sort keys. *Pos* has the form *m.n* optionally followed by one or more of the flags **bdfinr**, where *m* specifies a number of fields to skip, *n* a number of characters to skip further into the next field, and the flags specify a special ordering rule for the key. A missing *.n* is taken to be 0. **+***pos* denotes the beginning of the key; **−***pos* denotes the first position after the key (end of line by default). Later keys are compared only when all earlier keys compare equal.

When no tab character has been specified, a field consists of nonblanks and any preceding blanks. Under the **−b** flag, leading blanks are excluded from a field. When a tab character has been specified, fields are strings separated by tab characters.

Lines that otherwise compare equal are ordered with all bytes significant.

These flag arguments are also understood:

m Merge only, the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs, except under the merge flag **−m**.

u Suppress all but one in each set of contiguous equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

Examples. Print a list of all the distinct *roff*(I) commands in a given document:

```
grep "^\." document | sort −u +0 −0.3
```

Print the password file *passwd*(V) sorted by user id:

```
sort −t: +2n /etc/passwd
```

FILES

/tmp/stm???

NAME

spell – find spelling errors

SYNOPSIS

spell [**–v**] [**–1**] file ...

DESCRIPTION

Spell collects words from the named files, and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, or suffixes) from words in the spelling list are printed on the standard output. If no files are named, words are collected from the standard input.

Spell omits *nroff*(1), *troff*(1), *neqn*(1), and *eqn*(1) constructions from the input.

The process may take several minutes.

Under the **–v** flag, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated.

The **–1** option causes the final output to appear in a single column instead of three columns. The normal header and pagination is also suppressed.

The spelling list is based primarily on Kucera and Francis, *Computational Analysis of Present-Day English* and the Merriam Webster *New International Dictionary, 2nd edition.* Other sources include lists of chemical elements, states, countries, provinces, capital cities, major cities; given names from Kucera and Francis; the most common surnames from a large telephone book; common names from the index of *Fieldbook of Natural History* by E. L. Palmer and H. S. Fowler; selected names from *Bulfinch's Mythology;* Bell System Practices; Bell Laboratories technical papers and manuals; the *Federalist* papers; random literary fragments; etc.

If the file "/usr/dict/spellhist" is writable, *spell* accumulates copies of its output there.

FILES

/bin/deroff, /usr/lib/spell[0123]: programs
/usr/lib/w2006: list of common words for primary filtering
/usr/dict/spellinglist
/usr/dict/stoplist: likely misspellings (e.g. thier=thy–y+ier) that would otherwise pass
/usr/dict/spellhist

SEE ALSO

typo(1)

BUGS

The coverage of the spelling list is uneven; new installations will probably wish to monitor the output for a few months to gather local additions.

NAME

spline — interpolate smooth curve

SYNOPSIS

spline [option] ...

DESCRIPTION

Spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output (R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *plot*(I).

The following options are recognized, each as a separate argument.

a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

k The next argument is used as the constant k used in the boundary value computation

$$y_0'' = ky_1'', \quad y_n'' = ky_{n-1}''$$

is set by the next argument. By default $k = 0$.

n Space output points so that approximately n points occur between the lower and upper x limits, where n is the next argument. (Default $n = 100$.)

p Make output periodic, i.e. match derivatives at ends. First and last input values should normally agree.

x Next 1 (or 2) arguments are lower (and upper) x limits. Normally these limits are calculated from the data. Automatic abcissas start at the lower limit (default 0).

SEE ALSO

plot(I)

BUGS

A limit of 1000 input points is enforced silently.

NAME

split – split a file into pieces

SYNOPSIS

split –n [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if – is given in its stead, then the standard input file is used.

NAME

strip – remove symbols and relocation bits

SYNOPSIS

strip name ...

DESCRIPTION

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

The effect of *strip* is the same as use of the –s option of *ld*.

FILES

/tmp/stm? temporary file

SEE ALSO

ld(I), as(I), nm(I)

NAME

stty – set terminal options

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output terminal. With no argument, it reports the current settings of the options. The option strings are selected from the following set:

even	allow even parity
–even	disallow even parity
odd	allow odd parity
–odd	disallow odd parity
raw	raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
–raw	negate raw mode
cooked	same as '–raw'
–nl	allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl	accept only new-line to end lines
echo	echo back every character typed
–echo	do not echo characters
lcase	map upper case to lower case
–lcase	do not map case
–tabs	replace tabs by spaces when printing
tabs	preserve tabs
ek	reset erase and kill characters back to normal # and @.
erase *c*	set erase character to *c*.
kill *c*	set kill character to *c*.
cr0 cr1 cr2 cr3	select style of delay for carriage return (see *stty*(II))
nl0 nl1 nl2 nl3	select style of delay for linefeed (see *stty*(II))
tab0 tab1 tab2 tab3	select style of delay for tab (see *stty*(II))
ff0 ff1	select style of delay for form feed (see *stty*(II))
tty33	set all modes suitable for the Model 33 TELETYPE®
tty37	set all modes suitable for the Model 37 TELETYPE
vt05	set all modes suitable for Digital Equipment Corp. VT05 terminal
tn300	set all modes suitable for a General Electric TermiNet 300
ti700	set all modes suitable for Texas Instruments 700 series terminal
tek	set all modes suitable for Tektronix 4014 terminal
hup	hang up dataphone on last close.
–hup	do not hang up dataphone on last close.
0	hang up phone line immediately
50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb	Set terminal baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

SEE ALSO

stty(II)

NAME

su — become privileged user

SYNOPSIS

su [name]

DESCRIPTION

Su allows one to become the super-user, who has all sorts of marvelous (and correspondingly dangerous) powers. In order for *su* to do its magic, the user must supply a password. If the password is correct, *su* will execute the Shell with the user-id set to that of the super-user. To restore normal user-id privileges, type an end-of-file to the super-user Shell.

The password demanded is that of the entry "root" in the system's password file.

To remind the super-user of his responsibilities, the Shell substitutes "#" for its usual prompt ("%"). The ordinary user's command path search sequence does not apply to the super-user. The super-user gets "/bin", "/etc", and "/" instead (no current directory).

The optional argument allows logging in as *name* without logging off as yourself. That is, you get the powers and privileges, if any, of the user whose *login* name is *name*. In this case (unless you already are the super-user), *su* asks for that user's password, rather than for the super-user password.

FILES

/etc/passwd	system's password file

SEE ALSO

sh(I), pexec(III)

BUGS

If you are the super-user and invoke *su* with an invalid argument (i.e., a non-existent *login name*), you will get the powers and privileges, if any, of the user whose user-id is 255 (regardless of whether there actually is such a user).

NAME

sum – print checksum of a file

SYNOPSIS

sum [file] ...

DESCRIPTION

Sum sums the contents of the bytes (mod 2^16) of each *file* specified. *Sum* prints the file name, the number of whole or partial 512-byte disk blocks read, and the summed value of its bytes in decimal.

In practice, *sum* is often used to vertify that all of a special file can be read without error.

NAME

switch – shell multi-way branch command

SYNOPSIS

switch arg
: label1
commands...
breaksw
. . .
: labeln
commands...
breaksw
: default
commands...
endsw

DESCRIPTION

Switch searches forward in the input file for the first one of:

1. a label that pattern-matches *arg*. The pattern-matching used is that of the Shell in generating argument lists.
2. the label *default*.
3. a matching *endsw* command.

The Shell resumes reading commands from the next line after the location where the search stopped. Thus, *switch* supplies a 'case' or 'computed goto' statement similar to that of C. Because ':' is ignored by the Shell, several labels may occur in order, so that the same sequence of commands is executed for several different values of *arg*.

The *breaksw* command searches forward to the next unmatched *endsw*, and is normally used at the end of the sequence of commands following each label. It may be omitted to allow common code to be shared among label values. Several *breaksw* commands may be written on the same line to exit from that many levels of nested *switch*—*endsw* pairs.

The optional label *default* should be placed last, since *switch* always stops upon discovering it. The construct can be nested: any labels enclosed by a *switch-endsw* pair are ignored by an outer *switch*. The most common use of *switch* is to process 'flag' arguments in a shell procedure.

SEE ALSO

if(I), sh(I), while(I)

DIAGNOSTICS

switch: missing endsw
breaksw: missing endsw

BUGS

None of these commands should be hidden behind semicolons. Nested groups hidden behind *if* or *else* may also cause trouble.

NAME

sync — update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to insure file system integrity. See *sync(II)* for details.

SEE ALSO

sync(II)

NAME

tabs – set tabs on terminal

SYNOPSIS

tabs [tabspec] [+f] [+mn] [+ln] [+ttype] [+q]

DESCRIPTION

Tabs sets the tab stops on the user's terminal according to the tab specification *tabspec*, after clearing any previous settings. The user must of course be logged in on a terminal with remotely-settable hardware tabs, including the DASI450 (DIABLO 1620 or XEROX 1700), GSI300 (DTC300 or DASI300), DASI300S (DTC300S), HP2640B (HP2640A, HP2644A, HP2645A, etc.), TELETYPE® Model 40/2, and General Electric TermiNet terminals.

Users of TermiNet terminals should be aware that they behave in a different way than most other terminals for some tab settings; the first number in a list of tab settings becomes the *left margin* on a TermiNet terminal. Thus, any list of tab numbers whose first element is other than 1 causes a margin to be left by a TermiNet, but not by other terminals. A tab list beginning with 1 causes the same effect regardless of terminal type. It is also possible to set a left margin on the DASI450 and DASI300S, although in a different way.

Four types of tab specification are accepted for *tabspec:* 'canned', repetitive, arbitrary, and file. If no arguments are given, the default value is **–8**, i.e., UNIX 'standard' tabs. The lowest column number is 1 and the highest is 158. Note that for *tabs*, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0, e.g., the DASI300, DASI300S, and DASI450.

–*code* Gives the name of one of a set of 'canned' tabs. The legal codes and their meanings are as follows:

–a 1,10,16,36,72
Assembler, IBM S/370, first format

–a2 1,10,16,40,72
Assembler, IBM S/370, second format

–c 1,8,12,16,20,55
COBOL, normal format

–c2 1,6,10,14,49
COBOL compact format (columns 1–6 omitted). Using this code, the first typed character corresponds to card column 7, one space gets you to column 8, and a tab reaches column 12. In order to get *send(I)* to prepend the blanks at the beginning, files using this tab setup should include a format specification (see *fspec*(V)) as follows:
<:t–c2 m6 s66 d:>

–c3 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1–6 omitted), with more tabs than **–c2**. THIS IS THE RECOMMENDED FORMAT FOR COBOL. The appropriate format specification is:
<:t–c3 m6 s66 d:>

−f 1,7,11,15,19,23
FORTRAN

−p 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

−s 1,10,55
SNOBOL

−u 1,12,20,44
UNIVAC 1100 Assembler

In addition to these 'canned' formats, three other types exist:

−*n* A repetitive specification requests tabs at columns 1+*n*, 1+2**n*, etc. Note that such a setting leaves a left margin of *n* columns on TermiNet terminals *only*. Of particular importance is the value **−8**: this represents the UNIX 'standard' tab setting, and is the most likely tab setting to be found at a terminal. It is required for use with the *nroff*(1) **−h** option for high-speed output (about 10% speed increase). Another special case is the value **−0**, implying no tabs at all.

n1,n2.... The arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. The maximum tab value accepted is 158. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered identical.

−−*file* If the name of a file is given, *tabs* reads the first line of the file, searching for a format specification (see *fspec(V)*). If it finds one there, it sets the tab stops according to it, otherwise it sets them as **−8**. If an actual format specification is found in the file, it is printed at the terminal to remind the user what it is, unless the **+f** flag is also included to suppress this output. This type of specification may be used to make sure that a tabbed file is printed with correct tab settings, and would be used with the *pr(1)* command:

tabs −−file; **pr** file

Any of the following may be used also; if a given flag occurs more than once, the last value given takes effect.

+f If the **−−***file* type of tab specification is used and this option given, no tab specification is printed at the terminal, but the tab stops are set. This option is most useful when *tabs* is invoked from a shell procedure or another command, rather than directly from a terminal. This option has no effect unless a **−−***file* form of specification is used.

+l*n* The length argument gives the number of the rightmost column at which a tab will be set by a repetitive−style specification. The default value is 132, but may be different if the **+t** argument implies a more appropriate value for the specific type of terminal being used. When examining printed output obtained from another computer, it is helpful to have tabs across the entire width of the terminal. Maximum usable values of *n* are 118 (TermiNet), 132 (any DASI in 10-pitch mode), 158 (any DASI in 12-pitch mode), and 80 (HP2640). Although *tabs* will accept larger values without diagnostics, using them may cause a terminal (especially a DASI) to behave strangely.

+m*n* The margin argument may be used for TermiNet, DASI450, and DASI300S terminals. It causes all tabs to be moved over *n* columns by making column *n*+*1* the left margin. If

+m is given without a value of *n*, the value assumed is 10. For a TermiNet, the first value in the tab list should be 1, or the margin will move even further to the right.

To reset the left margin of a DASI450 to the normal(leftmost) position, type:

tabs +t450 +m0

The margin on a DASI450 or DASI300S is reset only when the **+m** flag is given explicitly. The margin is not settable on a DASI300, and is settable on the DASI450 and DASI300S *only* when the **+t** option is used (see below).

+q The ('quick') flag suppresses the emission of characters to clear previously set tabs. It can be used if the terminal is know to be clear already, i.e., just after it has been powered up or reset.

+t*type* The terminal type can be supplied to help *tabs* optimize its output for specific kinds of terminals, and is sometimes required when certain functions of some terminals are desired. This argument interacts with **+l** by setting different defaults for different terminals, and different maximum lengths. It interacts with **+m** because different methods must be used to set margins on the various terminals.

Given below are the possible cases for **+t** argument, listing argument value, maximum length, default length if **+l** is omitted, and notes. The notes give the following codes: 'S' for a short (several characters) clearing sequence, 'L' for a long sequence (about 60 characters), 'M' for a settable margin, and a list of the terminal types expected.

Value	Maximum	Default	S/L	M	Terminal(s)
+t300	158	132	L		GSI300 (DTC300 or DASI300)
+tgsi	"	"	"		"
+t300S	158	132	S	M	DASI300S (DTC300S)
+tgsis	"	"	"	"	"
+t450	158	132	S	M	DASI450 (DIABLO 1620 or XEROX 1700)
+t1620	"	"	"	"	"
+ttn	118	118	S	M	TermiNet 300 or 1200
+thp	80	80	L		HP2640A, HP2640B
+t40−2	80	80	S		TELETYPE 40/2
+t	158	132	S		any with settable tabs
omitted	158	132	L		any with settable tabs

Omitting the **+t** argument entirely will work for most situations. You should probably try to type the least that will work, and be more specific only when necessary.

Tab-setting is performed using the standard output.

DIAGNOSTICS

"illegal tabs" when arbitrary tabs are ordered incorrectly, or include any value greater than 158.
"illegal increment" when a zero or missing increment value is found in an arbitrary specification.
"unknown tab code" when a 'canned' code cannot be found.
"can't open" if −−file option used, and file can't be opened.
"file indirection" if −−file option used and the specification in that file points to yet another file. Indirection of this form is not permitted.

EXIT CODES

0 – normal
1 – for any error

SEE ALSO

fspec(V), nroff(I), reform(I), send(I)
GSI300(VII), DASI450(VII), HP2640(VII), TERMINET(VII)

BUGS

It is sad, but true, that it is often necessary to specify the terminal type. Various terminals use totally inconsistent ways of clearing tabs and setting margins. *Tabs* clears only 20 tabs (on terminals requiring a long sequence), but is willing to set 40.

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [±number[**lbc**]] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning, or –*number* from the end of the input. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b** or **c**. When no units are specified, counting is by lines.

SEE ALSO

dd(1)

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tbl – format tables for nroff or troff

SYNOPSIS

tbl [files] ...

DESCRIPTION

Tbl is an *nroff*(I) or *troff*(I) preprocessor for formatting tables. The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables, and which are reformatted. There are several global options; if any are desired, they are specified on the first line after .TS as a series of keywords separated by blanks or commas and followed by a semicolon. The possible words are:

center - center the table
expand - format the table to fill the current line length
box - enclose the table in a box
allbox - draw all possible lines so that every item is in a box

After this line (or after .TS if no global options are given) are the lines describing the table format. Each line describes a line of the actual table. One letter is used for each column. As many lines are as needed to describe the table are given; the last line should end with the character "." to signal the end of the format information. The last line of the description will apply to all following lines of the table. The legal characters to describe a column are:

c center within the column

r right-adjust

l left-adjust

n numerical adjustment: the units digits of numbers are aligned.

s span the previous entry over this column.

_ replace this entry with a horizontal line

= replace this entry with a double horizontal line

A column letter may be followed by an integer giving the number of spaces between this column and the next; 3 is default. A column letter may be preceded by a "|" character to indicate that a vertical line is to be drawn to the left of this column. Letting \t represent a tab (which must be typed as a genuine tab), the input:

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\t1644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields:

Town	Household Population Households	
	Number	Size
Bedminster	789	3.26
Bernards Twp.	3087	3.74
Bernardsville	2018	3.30
Bound Brook	3425	3.04
Branchburg	1644	3.49
Bridgewater	7897	3.81
Far Hills	240	3.19

If a table element contains only "_" or "=", a single or double line (respectively) is drawn across the *column* at that point. If a table line contains only "_" or "=", a single or double line (respectively) is drawn all the way across the *table*.

If a column describer contains the character "|", a vertical line is drawn to the left of that column beginning at the point in the column corresponding to the position of the vertical bar in the describer, and extending to the bottom of the table.

If no arguments are given, *tbl* reads the standard input, so it may be used as a filter. When it is used with *eqn* or *neqn*, the *tbl* command should be first, to minimize the volume of data passed through pipes.

SEE ALSO

TBL — A Program to Format Tables by M. E. Lesk.

NAME

tee – pipe fitting

SYNOPSIS

tee [name ...]

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the named files.

NAME

time – time a command

SYNOPSIS

time command

DESCRIPTION

The given command is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command.

The times are printed on the diagnostic output stream.

BUGS

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

tp – manipulate DECtape and magtape

SYNOPSIS

tp [key] [name ...]

DESCRIPTION

Tp saves and restores files on DECtape or magtape. Its actions are controlled by the *key* argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped, restored, or listed. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r The named files are written on the tape. If files with the same names already exist, they are replaced. 'Same' is determined by string comparison, so './abc' can never be the same as '/usr/dmr/abc' even if '/usr/dmr' is the current directory. If no file argument is given, '.' is the default.

u updates the tape. U is like **r**, but a file is replaced only if its modification date is later than the date stored on the tape; that is to say, if it has changed since it was dumped. U is the default command if none is given.

d deletes the named files from the tape. At least one name argument must be given. This function is not permitted on magtapes.

x extracts the named files from the tape to the file system. The owner and mode are restored. If no file argument is given, the entire contents of the tape are extracted.

t lists the names of the specified files. If no file argument is given, the entire contents of the tape is listed.

The following characters may be used in addition to the letter which selects the function desired.

m Specifies magtape as opposed to DECtape.

0,...,7 This modifier selects the drive on which the tape is mounted. For DECtape, 'x' is default; for magtape '0' is the default.

v Normally *tp* does its work silently. The **v** (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.

c means a fresh dump is being created; the tape directory is zeroed before beginning. Usable only with **r** and **u**. This option is assumed with magtape since it is impossible to selectively overwrite magtape.

f causes new entries on tape to be 'fake' in that no data is present for these entries. Such fake entries cannot be extracted. Usable only with **r** and **u**.

i Errors reading and writing the tape are noted, but no action is taken. Normally, errors cause a return to the command level.

w causes *tp* to pause before treating each file, type the indicative letter and the file name (as with **v**) and await the user's response. Response **y** means 'yes', so the file is treated. Null response means 'no', and the file does not take part in whatever is being done. Response **x** means 'exit'; the *tp* command terminates immediately. In the **x**

function, files previously asked about have been extracted already. With **r**, **u**, and **d** no change has been made to the tape.

FILES

/dev/tap?
/dev/mt?

DIAGNOSTICS

Several; the non-obvious one is 'Phase error', which means the file changed after it was selected for dumping but before it was dumped.

BUGS

A single file with several links to it is treated like several files.

NAME

tr – transliterate

SYNOPSIS

tr [**–cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. Any combination of the options **–cds** may be used: **–c** complements the set of characters in *string1* with respect to the universe of characters whose ascii codes are 001 through 377 octal; **–d** deletes all input characters in *string1;* **–s** squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[*a – b*] stands for the string of characters whose ascii codes run from character *a* to character *b*.

[*a***n*], where *n* is an integer or empty, stands for *n*-fold repetition of character *a*. *N* is taken to be octal or decimal according as its first digit is or is not zero. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character '\' may be used as in the Shell to remove special meaning from any character in a string. In addition, '\' followed by 1, 2, or 3 octal digits stands for the character whose ascii code is given by those digits.

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the Shell; 012 is the ascii code for newline.

```
tr –cs "[A–Z][a–z]" "[\012*]" <file1 >file2
```

SEE ALSO

sh(I), ed(I), ascii(V)

BUGS

Won't handle ascii NUL in *string1* or *string2;* always deletes NUL from input.

NAME

nroff, troff – text formatters

SYNOPSIS

nroff (or **troff**) [options] files

DESCRIPTION

NROFF and TROFF accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input. If no file names are given, input is taken from the standard input. The options, which may appear in any order so long as they appear before the filenames, are:

Option	*Effect*
–o*list*	Print only pages whose page numbers appear in *list*, which consists of numbers and number ranges separated by commas. A number range has the form *N*–*M* and means pages *N* through *M* inclusive; an initial –*N* means from the beginning to page *N*; and a final *N*– means from *N* to the end.
–n*N*	Number first generated page *N*.
–s*N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a new-line character. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow the changing of cassettes, and will resume after the phototypesetter START button is pressed.
–m*name*	Prepends the macro file **/usr/lib/tmac.***name* to the input files.
–r*aN*	Register *a* (one-character name) is set to *N*.
–i	Read standard input after the input files are exhausted.
–q	Invoke the simultaneous input-output mode of the **rd** request.
	NROFF Only
–T*type*	Specifies the output terminal type. Currently defined values for *type* are **37** for the (default) Model 37 TELETYPE®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300** for the DASI-300, **450** for the DASI-450 (or Diablo Hyterm) and **300S** for the DASI-300S. For 12-pitch, use **300-12**, **300S-12**, and **450-12**.
–e	Produce equally-spaced words in adjusted lines, using full terminal resolution.
–h	Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.
	TROFF Only
–t	Direct output to the standard output instead of the phototypesetter.
–f	Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-w Wait until phototypesetter is available, if it is currently busy.

-b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

-a Send a printable (ASCII) approximation of the results to the standard output.

-p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

-g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

FILES

/usr/lib/suftab suffix hyphenation tables
/tmp/ta00000 temporary file
/usr/lib/tmac.* standard macro files
/usr/lib/term/* (NROFF only) terminal driving tables
/usr/lib/font/* (TROFF only) font width tables

SEE ALSO

NROFF/TROFF User's Manual by J. F. Ossanna.
A TROFF Tutorial by B. W. Kernighan.
tbl(I).
For NROFF, see neqn(I), col(I), and tabs(I)
For TROFF, see eqn(I).

NAME

tty – get terminal name

SYNOPSIS

tty

DESCRIPTION

Tty gives the name of the user's terminal in the form 'tty*n*' for *n* a digit or letter. The actual path name is then '/dev/tty*n*'.

DIAGNOSTICS

'not a tty' if the standard input file is not a terminal.

NAME

typo – find possible typos

SYNOPSIS

typo [**–1**] [**–n**] file ...

DESCRIPTION

Typo hunts through a document for unusual words, typographic errors, and *hapax legomena* and prints them on the standard output.

The words used in the document are printed out in decreasing order of peculiarity along with an index of peculiarity. An index of 10 or more is considered peculiar. Printing of certain very common English words is suppressed.

The statistics for judging words are taken from the document itself, with some help from known statistics of English. The **–n** option suppresses the help from English and should be used if the document is written in, for example, Urdu.

The **–1** option causes the final output to appear in a single column instead of three columns. The normal header and pagination is also suppressed.

Roff(I) and *nroff*(I) control lines are ignored. Upper case is mapped into lower case. Quote marks, vertical bars, hyphens, and ampersands within words are equivalent to spaces. Words hyphenated across lines are put back together.

FILES

/tmp/ttmp??
/usr/lib/salt
/usr/lib/w2006

SEE ALSO

spell(I)

BUGS

Because of the mapping into lower case and the stripping of special characters, words may be hard to locate in the original text.

The escape sequences of *troff*(I) are not correctly recognized.

SEE ALSO

spell(I)

NAME

uname – print name of current UNIX

SYNOPSIS

uname

DESCRIPTION

Uname prints the current name of UNIX on the standard output file. It is mainly useful to determine what system one is using.

SEE ALSO

uname(II)

NAME

uniq – report repeated lines in a file

SYNOPSIS

uniq [**–udc** [+n] [–n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. Note that repeated lines must be adjacent in order to be found; see *sort*(I). If the **–u** flag is used, just the lines that are not repeated in the original file are output. The **–d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **–u** and **–d** mode outputs.

The **–c** option supersedes **–u** and **–d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

–*n* The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+*n* The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

sort(I), comm(I)

NAME

units – conversion program

SYNOPSIS

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

You have: inch
You want: cm
* *2.54000e+00*
/ *3.93701e−01*

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

You have: 15 pounds force/in2
You want: atm
* *1.02069e+00*
/ *9.79730e−01*

Units only does multiplicative scale changes. Thus it can convert Kelvin to Rankine, but not Centigrade to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter
c	speed of light
e	charge on an electron
g	acceleration of gravity
force	same as g
mole	Avogadro's number
water	pressure head per unit height of water
au	astronomical unit

'Pound' is a unit of mass. Compound names are run together, e.g. 'lightyear'. British units that differ from their US counterparts are prefixed thus: 'brgallon'. For a complete list of units, 'cat /usr/lib/units'.

FILES

/usr/lib/units

NAME

vp – Versatec print

SYNOPSIS

vp [**–b**bin] [**–o**offset] [**–t**tspec] [**–n**] [**–r**name] cmd [args]

DESCRIPTION

Vp builds a *sh*(I) command file in directory **/usr/vpd**, and invokes **/etc/vpd** (the Versatec daemon). The command file has the form:

```
: logname
vpbrk bin logname
chdir curdir
= p < logdir/.path
cmd args ^ reform tspec –0 +poffset
cat /usr/vpd/.X
[ echo shfile finished ^ mail logname >/dev/null ]
[ echo shfile finished ^ write logname >/dev/null ]
rm –f shfile
```

Here *logname* is your login name, *curdir* is your current directory when you executed *vp*, *logdir* is your login directory, *shfile* is the name that *vp* selects for the generated command file, *bin* is your data station bin (see below), *offset* is the offset for *reform*(I) (see below), and *cmd* and *args* are the command and optional arguments specified on the command line.

The Versatec daemon, */etc/vpd*, invokes *sh*(I) on the command files that *vp* queues up in */usr/vpd*. The daemon redirects the standard output of each command file to the Versatec printer.

The keyletter arguments are as follows:

- **–b** Your data station bin.
- **–o** The offset for *reform*. The default is 12.
- **–t** The first tabspec for *reform*. The default is '—'.
- **–n** A flag that includes the optional "mail" and "write" lines in the command file.
- **–r** The file named *name* is to be removed after printing is completed.

Example:

```
vp –bx123 pr –l84 myfile
```

OPERATIONS NOTE:

Execute **/etc/vpd** after replacing paper in the Versatec printer.

FILES

/usr/vpd/*	queued command files
/usr/vpd/.X	terminator
/usr/bin/vpbrk	break page generator
/dev/vp0	Versatec printer
/etc/vpd	daemon program

BUGS

There should be a *vp*(IV) and a *vpd*(VIII).
Only printers with DMA interfaces are handled; plotting is tolerated, but not supported.
You cannot pipe into *vp*.

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait until all processes started with & have completed, and report on abnormal terminations.

Because the *wait*(II) system call must be executed in the parent process, the Shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(I)

BUGS

After executing *wait* you are committed to waiting until termination, because interrupts and quits are ignored by all processes concerned. The only out, if the process does not terminate, is to *kill* it (see *kill* (I)) from another terminal or to hang up.

NAME

wc – word count

SYNOPSIS

wc [**–l**] [name ...]

DESCRIPTION

Wc counts lines and words in the named files, or in the standard input if no name appears. A word is a maximal string of printing characters delimited by spaces, tabs or newlines. All other characters are simply ignored.

The **–l** flag suppresses all output except the line count.

NAME

what – identify files

SYNOPSIS

what name ...

DESCRIPTION

What searches the given files for all occurrences of the pattern which *get*(I) substitutes for %Z% (this is @(#) at this printing) and prints out what follows until the first '"', '>', newline, or null character. For example, if the C program in file 'f.c' contains

```
char iden__[] "@(#)identification information";
```

and f.c is compiled to yield 'f.o' and 'a.out', then the command

```
what f.c f.o a.out
```

will print

```
f.c:
        identification information
f.o:
        identification information
a.out:
        identification information
```

What is intended to be used in conjunction with the SCCS command *get*(I), which automatically inserts identifying information, but it can also be used where the information is inserted manually.

SEE ALSO

get(I), help(I)

DIAGNOSTICS

Use *help*(I) for explanations.

BUGS

It's possible that an unintended occurrence of the pattern @(#) could be found just by chance, but this causes no harm in nearly all cases.

NAME

whatsnew – compare file modification dates

SYNOPSIS

whatsnew [–yymmdd] [listfile]

DESCRIPTION

Whatsnew will compare the modification dates of files listed in *listfile* against the date supplied and report those files that have changed since that date. By default, the modification time of the file *.newdate* in the user's login directory will be used as the comparison date. Similarly, the file *.newlist* in the user's login directory will be used if *listfile* is omitted.

If a date is not supplied and *.newlist* exists, it will be re-created. This will essentially update the default comparison date used by subsequent *whatsnew* commands.

Entries in the list file should be relative to the login directory, one per line. If an entry is a directory, files in that directory will be compared. Only one level of directory searching is performed.

FILES

/logindir/.newlist
/logindir/.newdate

DIAGNOSTICS

"bad date" if the supplied date is earlier than 1970.
"cannot read list" if the list file is not readable.
"cannot access file status" if it can't.

NAME

while – shell iteration command

SYNOPSIS

while expr

commands... (may include *break* or *continue*)

end

DESCRIPTION

While evaluates the expression *expr*, which is similar to (and a superset of) the expression described in *if(I)*. If the expression is true, *while* does nothing, permitting the command(s) on following lines to be read and executed by the Shell. If the expression is false, the input file is effectively searched for the matching *end* command, and the Shell resumes execution of the command(s) on the line following the *end*. The *while-end* grouping may be nested up to three levels deep.

In addition to the type of expression permitted by *if*, *while* treats a single, nonnull argument as a true expression, and treats a single null argument or lack of arguments as a false expression.

The *break* command terminates the nearest enclosing *while-end* group, causing execution to resume after the nearest succeeding unmatched *end*. Exit from *n* levels is obtained by writing *n break* commands on the same line.

The *continue* command causes execution to resume at a preceding *while*, i.e., the *while* that begins the smallest loop containing the *continue*.

A common loop is that of processing arguments one at a time: see *shift(I)*.

The following is a shell procedure that is also a filter. It reads a line at a time from the standard input that existed when the procedure was invoked, exiting on end-of-file.

```
while 1
        = a <— || exit
        commands using $a ...
end
```

SEE ALSO

goto(I), if(I), onintr(I), sh(I), shift(I), switch(I)

DIAGNOSTICS

while: missing end
while: >3 levels
while: syntax errors like those of if.
break: missing end
break: used outside loop
continue: used outside loop
end: used outside loop

BUGS

A *goto* may be used to terminate one or more *while-end* groupings. Those who use it to branch into a loop will receive appropriately peculiar results. When an interrupt is caught and transfer to a label caused by use of *onintr(I)*, all currently effective *while-end* loops are cancelled, i.e., the *onintr* performs a *goto* that breaks all loops. Neither *while* nor *end* may be hidden behind semicolons or used within other commands.

NAME

who – who is on the system

SYNOPSIS

who [who-file] [**am I**]

DESCRIPTION

Who, without an argument, lists the name, terminal channel, and login time for each current UNIX user.

Without an argument, *who* examines the **/etc/utmp** file to obtain its information. If a file is given, that file is examined. Typically the given file will be **/usr/adm/wtmp,** which contains a record of all the logins since it was created. Then *who* lists logins, logouts, and crashes since the creation of the wtmp file. Each login is listed with user name, terminal name (with '/dev/' suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with 'x' in the place of the device name, and a fossil time indicative of when the system went down.

With two arguments, *who* behaves as if it had no arguments except for restricting the printout to the line for the current terminal. Thus 'who am I' (and also 'who are you') tells you who you are logged in as.

FILES

/etc/utmp

SEE ALSO

login(I), init(VIII)

NAME

write — write to another user

SYNOPSIS

write user [ttyno]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message

message from yourname ...

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal or an interrupt is sent. At that point *write* writes 'EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *ttyno* argument may be used to indicate the last character of the appropriate terminal name.

Permission to write may be denied or granted by use of the *mesg*(I) command. At the outset writing is allowed. Certain commands, in particular *nroff* and *pr*, disallow messages in order to prevent messy output.

If the character '!' is found at the beginning of a line, *write* calls the Shell to execute the rest of the line as a command.

The following protocol is suggested for using *write:* when you first write to another user, wait for that user to write back before starting to send. Each party should end each message with a distinctive signal (**(o)** for 'over' is conventional) that the other may reply. **(oo)** (for 'over and out') is suggested when conversation is about to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute '!'

SEE ALSO

mesg(I), who(I), mail(I)

NAME

xargs – construct argument list(s) and execute command

SYNOPSIS

xargs [flags] [command [initial-args]]

DESCRIPTION

Xargs combines the fixed *initial-args* with args read from standard input to execute the specified *command* one or more times. The *command* can either be executed for each line of args read, with all args read for each automatically-determined group of (at most *size* characters of) args read, or for each user-specifiable *number* of args read.

Specifically, *xargs* reads the standard input for arguments, using them to construct one or more arg lists with *initial-args* (if any), and executes *command* with each such constructed argument list; the directory containing *command*, which may also be a Shell file, must be in one's *.path* file. If *command* is omitted, **/bin/echo** is used. Excepting the use of the insert option (**–i** flag, see below), arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or newlines; however, quoted strings (including embedded blanks or tabs) may also form all or part of an argument.

Excepting the **–i** option, each argument list will be constructed starting with the *initial-args*, followed by an appropriate number of arguments read from standard input. Flags **–i**, **–l**, and **–n** modify how args are selected for each command invocation; when none of these flags are coded, each arg list is built from the continuously-read args from standard input, up to *size* characters per list maximum, until there are no more args. When there are flag conflicts (e.g., **–l** vs. **–n**), the last flag has precedence. Flag values are:

–x Causes *xargs* to terminate if any arg list would be greater than *size* characters; **–x** is forced by the options **–i** and **–l**. When neither of the options **–i**, **–l**, or **–n** are coded, the total length of all args must be within the *size* limit.

–l *Command* is executed for each non-null line of args from standard input. A line is considered to end with the first newline *unless* the last character of the line is a blank or a tab; in either of these cases, the blank/tab signals continuation through the next non-null line. Option **–x** is forced.

–ireplstr Insert mode: *command* is executed for each line from standard input, taking the entire line as one entity, inserting it in *initial-args* for each occurrence of *replstr*. A maximum of 5 args in *initial-args* may each contain one or more instances of *replstr*. Blanks and tabs at the beginning of each line are thrown away, as are empty lines. Constructed args may not grow larger than 255 characters, and option **–x** is also forced. '{}' is assumed for *replstr* if not specified.

–nnumber Execute *command* using as many standard input args as possible, up to *number* args maximum. Fewer args will be used if their total size is greater than *size* characters, and for the last invocation if there are fewer than *number* args remaining. If option **–x** is also coded, each *number* args must fit in the *size* limitation, else *xargs* terminates execution.

–t Trace mode: the *command* and each constructed arg list are echoed to file descriptor 2 just prior to their execution.

–p Prompt mode: the user is asked whether to execute *command* each invocation. Trace mode (**–t**) is turned on to print the command instance to be executed, followed by the prompt '?...'. A reply of **y** (optionally followed by anything)

will execute the command; anything else, including just a carriage return, skips that particular invocation of *command*.

−ssize The maximum total size of each arg list is set to *size* characters; *size* must be a positive integer less than or equal to 470. If −s is not coded, 470 is taken as the default. Note that the character count for *size* includes one extra character for each arg and the count of characters in the command name.

−eeofstr *Eofstr* is taken as the logical end-of-file string. Underbar (_) is assumed for the logical EOF string if −e is not coded. −e with no *eofstr* coded turns off the logical EOF string capability (underbar is taken literally). *Xargs* reads standard input until either end-of-file or the logical EOF string is encountered.

In args read from standard input, characters may be escaped (by a '\') outside of quoted strings; quoted strings are stripped of the delimiting quotes, with the contents taken literally.

Xargs will terminate if either it receives a return code of minus one from, or if it cannot execute, *command*.

EXAMPLES

The following will copy all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs −i −t mv $1/{} $2/{}
```

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the file *log:*

```
(logname; date; echo $0 $*) | xargs >>log
```

The user is asked which files in the current directory are to be archived and archives them into *arch* (1.) one at a time, or (2.) many at a time.

```
1. ls | xargs −p −l ar r arch
2. ls | xargs −p −l | xargs ar r arch
```

The following will execute *com* with successive pairs of args originally typed as Shell arguments:

```
echo $* | xargs −n2 com
```

DIAGNOSTICS

arg list too long
command not executed or returned −1
Missing quote? <string>
too many args with *replstr*
insert-buffer overflow
max arg size with insertion via *replstr* exceeded
unknown option: <option>
0 < max-line-size <= 470: <−s option as coded>
#args must be positive int: <−n option as coded>
can't read from tty for −p

NAME

yacc – yet another compiler-compiler

SYNOPSIS

yacc [**–vrd**] [grammar]

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output is *y.tab.c*, which must be compiled by the C compiler and loaded with any other routines required (perhaps a lexical analyzer) and the *yacc* library:

```
cc y.tab.c other.o –ly
```

If the **–v** flag is given, the file *y.output* is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

The **–r** flag causes *yacc* to accept grammars with Ratfor actions, and produce Ratfor output on *y.tab.r*. Typical usage is then

```
rc y.tab.r other.o
```

If the **–d** flag is used, the file *y.tab.h* is generated with the *define* statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than *y.tab.c* to access the token codes.

SEE ALSO

lex(I)
LR Parsing by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974.
YACC – Yet Another Compiler Compiler by S. C. Johnson.

FILES

y.output	
y.tab.c	
y.tab.r	when ratfor output is obtained
y.tab.h	defines for token names
yacc.tmp, yacc.acts	temporary files
/lib/liby.a	runtime library for compiler
/usr/lib/yaccopar	parser prototype for C programs
/usr/lib/yaccrpar	parser prototype for Ratfor programs

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the *y.output* file.

BUGS

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

INTRODUCTION TO SYSTEM CALLS

Section II of this manual lists all the entries into the system. In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. The presence of an error is most easily tested by the instructions *bes* and *bec* ("branch on error set (or clear)"). These are synonyms for the *bcs* and *bcc* instructions.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is −1; the individual sections specify the details.

In both cases an error number is also available. In assembly language, this number is returned in r0 on erroneous calls. From C, the external variable *errno* is set to the error number. *Errno* is not cleared on successful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror(III)*.

The possible error numbers are not recited with each writeup in section II, since many errors are possible for most of the calls. Here is a list of the error numbers, their names inside the system (for the benefit of system-readers), and the messages available using *perror*. A short explanation is also provided.

0 – (unused)

1 EPERM Not owner and not super-user
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
The process whose number was given to *signal* does not exist, or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error occurred during a *read* or *write*. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialed in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than the maximum allowable (counting the null at the end of each argument) is presented to *exec*. The maximum is a configuration dependent parameter.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with one of the magic numbers 407, 410, or 411.

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).

10 ECHILD No children
Wait and the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a *fork*, the system's process table is full and no more processes can for the moment be created.

12 ENOMEM Not enough core
During an *exec* or *break*, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments is such as to require more than the existing 8 segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Memory fault
A memory fault occurred while passing data between the user and the system. Most likely the result of bad arguments to the system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g., in *mount*.

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an open file or some process's current directory.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link*.

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir*.

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: currently, dismounting a non-mounted device, mentioning an unknown signal in *signal*, and giving an unknown request in *stty* to the TIU special file.

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
Only 15 files can be open per process.

25 ENOTTY Not a terminal
The file mentioned in *stty* or *gtty* is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
An attempt to make a file larger than the maximum of 32768 blocks.

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Seek on pipe
A *seek* was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 127 links to a file.

32 EPIPE Write on broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

NAME

access – determine accessibility of file

SYNOPSIS

(access = 33.)
sys access; name; mode

access(name, mode)
char *name;

DESCRIPTION

Access checks the given file *name* for accessibility according to *mode*, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. An appropriate error indication is returned if one or more of the desired access modes would not be granted.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by *access*, but an attempt to open it for writing will fail (although files may be created there); a file may look excutable, but *exec*(II) will fail unless it is in proper format.

SEE ALSO

stat(II)

DIAGNOSTICS

C-bit is set on disallowed accesses, and the error code is in r0; from C, −1 is returned and the error code is in *errno*. 0 is returned from successful tests.

NAME

alarm – schedule signal after specified time

SYNOPSIS

(alarm = 27.)
(seconds in r0)
sys **alarm**

alarm(seconds)
int seconds;

DESCRIPTION

Alarm causes signal number 14 to be sent to the invoking process in a number of seconds given by the argument. Unless caught, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is cancelled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 65535 seconds. The old value of the alarm clock is returned in r0. In C, that value is returned.

SEE ALSO

pause(II), sleep(III)

NAME

break, brk, sbrk – change core allocation

SYNOPSIS

(break = 17.)
sys break; addr

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Break sets the system's idea of the lowest location not used by the program (called the break) to *addr* (rounded up to the next multiple of 64 bytes). Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

From C, *brk* will set the break to *addr*. The old break is returned.

In the alternate entry *sbrk*, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *break*.

SEE ALSO

exec(II), alloc(III), end(III)

DIAGNOSTICS

The c-bit is set if the program requests more memory than the system limit or if more than 8 segmentation registers would be required to implement the break. From C, −1 is returned for these errors.

BUGS

Setting the break in the range 0177700 to 0177777 is the same as setting it to zero.

NAME

chdir – change working directory

SYNOPSIS

(chdir = 12.)
sys chdir; dirname

chdir(dirname)
char *dirname;

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. *Chdir* causes this directory to become the current working directory.

SEE ALSO

chdir(I)

DIAGNOSTICS

The error bit (c-bit) is set if the given name is not that of a directory or is not readable. From C, a −1 returned value indicates an error, 0 indicates success.

NAME

chmod – change mode of file

SYNOPSIS

(chmod = 15.)
sys chmod; name; mode

chmod(name, mode)
char *name;

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode*. Modes are constructed by ORing together some combination of the following:

4000 set user ID on execution
2000 set group ID on execution
1000 save text image after execution
0400 read by owner
0200 write by owner
0100 execute (search on directory) by owner
0070 read, write, execute (search) by group
0007 read, write, execute (search) by others

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

SEE ALSO

chmod(I)

DIAGNOSTIC

Error bit (c-bit) set if *name* cannot be found or if current user is neither the owner of the file nor the super-user. From C, a −1 returned value indicates an error, 0 indicates success.

NAME

chown – change owner and group of a file

SYNOPSIS

(chown = 16.)
sys chown; name; owner

chown(name, owner)
char *name;

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its owner and group changed to the low and high bytes of *owner* respectively.

SEE ALSO

chown(I), chgrp(I), passwd(V)

DIAGNOSTICS

The error bit (c-bit) is set on illegal owner changes. From C, a −1 returned value indicates error, 0 indicates success.

NAME

close – close a file

SYNOPSIS

(close = 6.)
(file descriptor in r0)
sys close

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an *open, creat,* or *pipe* call, *close* closes the associated file. A close of all files is automatic on *exit,* but since processes are limited to 15 simultaneously open files, *close* is necessary for programs which deal with many files.

SEE ALSO

creat(II), open(II), pipe(II)

DIAGNOSTICS

The error bit (c-bit) is set for an unknown file descriptor. From C, a −1 indicates an error, 0 indicates success.

NAME

creat — create a new file

SYNOPSIS

(creat = 8.)
sys creat; name; mode
(file descriptor in r0)

creat(name, mode)
char *name;

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*. See *chmod(II)* for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned (in r0).

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat*, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write(II), close(II), stat(II)

DIAGNOSTICS

The error bit (c-bit) may be set if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

From C, a −1 return indicates an error.

NAME

csw – read console switches

SYNOPSIS

```
(csw = 38.)
sys     csw

getcsw( )
```

DESCRIPTION

The setting of the console switches is returned (in r0).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

(dup = 41.)
(file descriptor in r0)
sys dup

dup(fildes)
int fildes;

DESCRIPTION

Given a file descriptor returned from an *open, pipe,* or *creat* call, *dup* will allocate another file descriptor synonymous with the original. The new file descriptor is returned in r0.

Dup is used more to reassign the value of file descriptors than to genuinely duplicate a file descriptor. Since the algorithm to allocate file descriptors returns the lowest available value, combinations of *dup* and *close* can be used to manipulate file descriptors in a general way. This is handy for manipulating standard input and/or standard output.

SEE ALSO

creat(II), open(II), close(II), pipe(II)

DIAGNOSTICS

The error bit (c-bit) is set if: the given file descriptor is invalid; there are already too many open files. From C, a −1 returned value indicates an error.

NAME

exec, execl, execv – execute a file

SYNOPSIS

```
(exec = 11.)
sys exec; name; args
...
name: <...\0>
...
args: arg0; arg1; ...; 0
arg0: <...\0>
arg1: <...\0>
  ...

execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name;
char *argv[ ];
```

DESCRIPTION

Exec overlays the calling process with the named file, then transfers to the beginning of the core image of the file. There can be no return from the file; the calling core image is lost.

Files remain open across *exec* calls. Ignored signals remain ignored across *exec*, but signals that are caught are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. *Exec* changes the effective user and group ID to the owner of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The real user ID is not affected.

The form of this call differs somewhat depending on whether it is called from assembly language or C; see below for the C version.

The first argument to *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once the called file starts execution, the arguments are available as follows. The stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings. The arguments are placed as high as possible in core.

```
sp→  nargs
     arg0
     ...
     argn
     -1

arg0:  <arg0\0>
       ...
argn:  <argn\0>
```

From C, two interfaces are available. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments; as in the basic call, the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance; the arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv)
int argc;
char **argv;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is not directly usable in another *execv*, since *argv[argc]* is −1 and not 0.

SEE ALSO

fork(II), pexec(III)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410, or 411 octal as first word), if maximum memory is exceeded, or if the arguments require more than 5120 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1.

BUGS

Only 5120 characters of arguments are allowed.

NAME

exit – terminate process

SYNOPSIS

(exit = 1.)
(status in r0)
sys exit

exit(status)
int status;

DESCRIPTION

Exit is the normal means of terminating a process. *Exit* closes all open files of the process, and notifies the parent process, if it is executing a *wait*. The low byte of r0 (resp. the argument to *exit*) is available as status to the parent process.

This call can never return.

SEE ALSO

wait(II)

NAME

fork – spawn new process

SYNOPSIS

(fork = 2.)
sys fork
(new process return)
(old process return)

fork( )

DESCRIPTION

Fork is the only way new processes are created. The new process's core image is a copy of that of the caller of *fork*. The only distinction is the return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process. This process ID is used by *wait*.

The two returning processes share all open files that existed before the call. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

From C, the child process receives a 0 return, and the parent receives a non-zero number which is the process ID of the child; a return of −1 indicates inability to create a new process.

SEE ALSO

wait(II), exec(II), pexec(III)

DIAGNOSTICS

The error bit (c-bit) is set in the old process if a new process could not be created because of lack of process space. From C, a return of −1 indicates an error.

NAME

fstat – get status of open file

SYNOPSIS

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf

fstat(fildes, buf)
struct inode *buf;

DESCRIPTION

This call is identical to *stat*, except that it operates on open files instead of files given by name. It is most often used to get the status of the standard input and output files, whose names are unknown.

SEE ALSO

stat(II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor is unknown; from C, a −1 return indicates an error, 0 indicates success.

NAME

getgid – get group identifications

SYNOPSIS

(getgid = 47.)
sys getgid

getgid( )

DESCRIPTION

Getgid returns a word (in r0), the low byte of which contains the real group ID of the current process. The high byte contains the effective group ID of the current process. The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set group ID" mode, to find out who invoked them.

SEE ALSO

setgid(II)

NAME

getpid – get process identification

SYNOPSIS

(getpid = 20.)
sys getpid
(pid in r0)

getpid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

NAME

getuid — get user identifications

SYNOPSIS

(getuid = 24.)
sys getuid

getuid( )

DESCRIPTION

Getuid returns a word (in r0), the low byte of which contains the real user ID of the current process. The high byte contains the effective user ID of the current process. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set user ID" mode, to find out who invoked them.

SEE ALSO

setuid(II)

NAME

gtty – get terminal status

SYNOPSIS

```
(gtty = 32.)
(file descriptor in r0)
sys gtty; arg
...
arg: .=.+6

gtty(fildes, arg)
int arg[3];
```

DESCRIPTION

Gtty stores in the three words addressed by *arg* the status of the terminal whose file descriptor is given in r0 (resp. given as the first argument). The format is the same as that passed by *stty*.

SEE ALSO

stty(II)

DIAGNOSTICS

Error bit (c-bit) is set if the file descriptor does not refer to a terminal. From C, a −1 value is returned for an error, 0, for a successful call.

NAME

indir – indirect system call

SYNOPSIS

(indir = 0.)
sys indir; syscall

DESCRIPTION

The system call at the location *syscall* is executed. Execution resumes after the *indir* call.

The main purpose of *indir* is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If *indir* is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, the executing process will get a fault.

NAME

kill – send signal to a process

SYNOPSIS

(kill = 37.)
(process number in r0)
sys kill; sig

kill(pid, sig);

DESCRIPTION

Kill sends the signal *sig* to the process specified by the process number in r0. See *signal*(II) for a list of signals.

The sending and receiving processes must have the same effective user ID. The super-user can kill any process.

If the process number is 0, the signal is sent to all processes which have the same process group number as the sender. If the process number is less than 0, the signal is sent to all processes for which the sender has permission. In both of the above cases, process 0 and process 1 are excluded. Note, process 0 is really the scheduler, and process numbers must be positive to avoid confusion with error indications in C.

SEE ALSO

kill(I), signal(II)

DIAGNOSTICS

The error bit (c-bit) is set if the process does not have permission, or if the process does not exist. From C, a −1 return indicates an error.

NAME

link – link to a file

SYNOPSIS

(link = 9.)

sys link; name1; name2

link(name1, name2)
char *name1, *name2;

DESCRIPTION

A link to *name1* is created; the link has the name *name2*. Either name may be an arbitrary path name.

SEE ALSO

ln(I), unlink(II)

DIAGNOSTICS

The error bit (c-bit) is set when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when more than 127 links are made. From C, a −1 return indicates an error, a 0 return indicates success.

NAME

logname, logdir, logtty, logpost – login information

SYNOPSIS

```
char    *logname(), *logdir(), *logtty();
logpost(buf)
char    *buf;
```

DESCRIPTION

Logname returns a pointer to the null-terminated login name (fits in 8 characters).

Logdir returns a pointer to the null-terminated login directory pathname (fits in 22 char).

Logtty returns a pointer to the tty letter.

These data are created by *login*(I) using the function *logpost* which is executable only by the super-user.

This function is kept in the **−lPW** library.

SEE ALSO

login(I), udata(II)

DIAGNOSTICS

Same as for *udata*(II).

NAME

mknod – make a directory or a special file

SYNOPSIS

(mknod = 14.)

sys mknod; name; mode; addr

mknod(name, mode, addr)
char *name;

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. The first physical address of the file is initialized from *addr*. Note that in the case of a directory, *addr* should be zero. In the case of a special file, *addr* specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(I), mknod(VIII), fs(V)

DIAGNOSTICS

Error bit (c-bit) is set if the file already exists or if the user is not the super-user. From C, a −1 value indicates an error.

NAME

mount – mount file system

SYNOPSIS

(mount = 21.)
sys mount; special; name; rwflag

mount(special, name, rwflag)
char *special, *name;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special;* from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

Only the super-user can execute mount.

SEE ALSO

mount(VIII), umount(II)

DIAGNOSTICS

Error bit (c-bit) set if: *special* is inaccessible or not an appropriate file; *name* does not exist; *special* is already mounted; *name* is in use; there are already too many file systems mounted.

NAME

nice – set program priority

SYNOPSIS

(nice = 34.)
(priority in r0)
sys nice

nice(priority)

DESCRIPTION

The scheduling *priority* of the process is changed to the argument. Positive priorities get less service than normal; 0 is default. Only the super-user may specify a negative priority. The valid range of *priority* is 20 to −128. The value of 4 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0.

The actual running priority of a process is the *priority* argument plus a number that ranges from 100 to 127 depending on the cpu usage of the process.

SEE ALSO

nice(I)

DIAGNOSTICS

The error bit (c-bit) is set if the user requests a *priority* outside the range of 0 to 20 and is not the super-user.

NAME

open – open for reading or writing

SYNOPSIS

(open = 5.)
sys open; name; mode
(file descriptor in r0)

open(name, mode)
char *name;

DESCRIPTION

Open opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The returned file descriptor should be saved for subsequent calls to *read, write,* and *close.*

SEE ALSO

creat(II), read(II), write(II), close(II)

DIAGNOSTICS

The error bit (c-bit) is set if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open. From C, a −1 value is returned on an error.

NAME

pause – indefinite wait

SYNOPSIS

(pause = 29.)

```
sys     pause

pause ();
```

DESCRIPTION

Pause causes its caller to suspend execution indefinitely. A caught signal is processed normally. The most plausible use of *pause* is in conjunction with an alarm-clock signal: *alarm(II)*.

SEE ALSO

alarm(II), signal(II)

NAME

pipe – create an interprocess channel

SYNOPSIS

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

pipe(fildes)
int fildes[2];

DESCRIPTION

The *pipe* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor returned in r1 (resp. fildes[1]), up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor returned in r0 (resp. fildes[0]) will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork* calls) will pass data through the pipe with *read* and *write* calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) return an end-of-file. Write calls under similar conditions generate a fatal signal (*signal*(II)); if the signal is ignored, an error is returned on the write.

SEE ALSO

sh(I), read(II), write(II), fork(II)

DIAGNOSTICS

The error bit (c-bit) is set if too many files are already open. From C, a −1 returned value indicates an error. A signal is generated if a write on a pipe with only one end is attempted.

NAME

profil – execution time profile

SYNOPSIS

(profil = 44.)
sys profil; buff; bufsiz; offset; scale

profil(buff, bufsiz, offset, scale)
char buff[ ];
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 177777(8) gives a 1-1 mapping of pc's to words in *buff*; 77777(8) maps each pair of instruction words together. 2(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is also turned off when an *exec* is executed but remains on in child and parent both after a *fork*.

SEE ALSO

monitor(III), prof(I)

NAME

ptrace — process trace

SYNOPSIS

(ptrace = 26.)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)

ptrace(request, pid, addr, data);

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging, but it should be adaptable for simulation of non-UNIX environments. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt." See *signal*(II) for the list. Then the traced process enters a stopped state and its parent is notified via *wait(II)*. When the child is in the stopped state; its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned (in r0). Request 1 indicates the instruction space, 2 indicates the data space. *addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even and in the data area. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. Request 4 specifies instruction space, 5 specifies data space. Attempts to write in pure procedure result in termination of the child, instead of going through or causing an error for the parent.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The *data* argument is taken as a signal number and the child's execution continues as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal which caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop.

8 The traced process terminates.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the

"termination" status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec(II)* calls.

SEE ALSO

wait(II), signal(II), cdb(I)

DIAGNOSTICS

From assembler, the c-bit (error bit) is set on errors; from C, −1 is returned and *errno* has the error code.

BUGS

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged.

Also, it should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

NAME

read – read from file

SYNOPSIS

(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes

read(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a terminal at most one line will be returned. In any event the number of characters read is returned (in r0).

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open(II), creat(II), dup(II), pipe(II)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the error bit (c-bit) is set. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes,* file descriptor not that of an input file. From C, a −1 return indicates the error.

NAME

seek – move read/write pointer

SYNOPSIS

(seek = 19.)
(file descriptor in r0)
sys seek; offset; ptrname

seek(fildes, offset, ptrname)

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

if *ptrname* is 0, the pointer is set to *offset*.

if *ptrname* is 1, the pointer is set to its current location plus *offset*.

if *ptrname* is 2, the pointer is set to the size of the file plus *offset*.

if *ptrname* is 3, 4 or 5, the meaning is as above for 0, 1 and 2 except that the offset is multiplied by 512.

If *ptrname* is 0 or 3, *offset* is unsigned, otherwise it is signed.

SEE ALSO

open(II), creat(II), tell(II)

DIAGNOSTICS

The error bit (c-bit) is set for an undefined file descriptor. From C, a −1 return indicates an error.

NAME

setgid – set process group ID

SYNOPSIS

(setgid = 46.)
(group ID in r0)
sys setgid

setgid(gid)

DESCRIPTION

The group ID of the current process is set to the argument. Both the effective and the real group ID are set. This call is only permitted to the super-user or if the argument is the real group ID.

SEE ALSO

getgid(II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

NAME

setuid – set process user ID

SYNOPSIS

(setuid = 23.)
(user ID in r0)
sys setuid

setuid(uid)

DESCRIPTION

The user ID of the current process is set to the argument. Both the effective and the real user ID are set. This call is only permitted to the super-user or if the argument is the real user ID.

SEE ALSO

getuid(II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

NAME

setpgrp – set process group number

SYNOPSIS

(setpgrp = 39.; not in assembler)
sys setpgrp

setpgrp()

DESCRIPTION

Setpgrp sets the process group number of the process to the process ID of the process. The process ID is guaranteed to be unique among the current process IDs and process group numbers, so that the new process group number will be unique. Process group numbers are used to group processes for catching signals.

SEE ALSO

kill(II), signal(II)

The value of the call is the old action defined for the signal.

After a *fork(II)* the child inherits all signals. *Exec(II)* resets all caught signals to default action.

SEE ALSO

kill(I), kill(II), ptrace(II), reset(III)

DIAGNOSTICS

The error bit (c-bit) is set if the given signal is out of range. In C, a −1 indicates an error.

The value of the call is the old action defined for the signal.

After a *fork(II)* the child inherits all signals. *Exec(II)* resets all caught signals to default action.

SEE ALSO

kill(I), kill(II), ptrace(II), reset(III)

DIAGNOSTICS

The error bit (c-bit) is set if the given signal is out of range. In C, a −1 indicates an error.

NAME

stat – get file status

SYNOPSIS

(stat = 18.)
sys stat; name; buf

stat(name, buf)
char *name;
struct inode *buf;

DESCRIPTION

Name points to a null-terminated string naming a file; *buf* is the address of a 36(10) byte buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be readable. After *stat, buf* has the following structure (starting offset given in bytes):

```
struct inode {
        char    minor;          /* +0: minor device of i-node */
        char    major;          /* +1: major device */
        int     inumber;        /* +2 */
        int     flags;          /* +4: see below */
        char    nlinks;         /* +6: number of links to file */
        char    uid;            /* +7: user ID of owner */
        char    gid;            /* +8: group ID of owner */
        char    size0;          /* +9: high byte of 24-bit size */
        int     size1;          /* +10: low word of 24-bit size */
        int     addr[8];        /* +12: block numbers or device number */
        int     actime[2];      /* +28: time of last access */
        int     modtime[2];     /* +32: time of last modification */
};
```

The flags are as follows:

```
100000  i-node is allocated
060000  2-bit file type:
    000000  plain file
    040000  directory
    020000  character-type special file
    060000  block-type special file.
010000  large file
004000  set user-ID on execution
002000  set group-ID on execution
001000  save text image after execution
000400  read (owner)
000200  write (owner)
000100  execute (owner)
000070  read, write, execute (group)
000007  read, write, execute (others)
```

SEE ALSO

ls(I), fstat(II), fs(V)

DIAGNOSTICS

Error bit (c-bit) is set if the file cannot be found. From C, a −1 return indicates an error.

NAME

stime – set time

SYNOPSIS

(stime = 25.)
(time in r0-r1)
sys stime

stime(tbuf)
int tbuf[2];

DESCRIPTION

Stime sets the system's idea of the time and date. Time is measured in seconds from 0000 GMT Jan 1, 1970. Only the super-user may use this call.

SEE ALSO

date(I), time(II), ctime(III)

DIAGNOSTICS

Error bit (c-bit) set if user is not the super-user.

NAME

stty – set mode of terminal

SYNOPSIS

```
(stty = 31.)
(file descriptor in r0)
sys stty; arg
...
arg:  .byte ispeed, ospeed; .byte erase, kill; mode

stty(fildes, arg)
struct {
        char    ispeed, ospeed;
        char    erase, kill;
        int     mode;
} *arg;
```

DESCRIPTION

Stty sets mode bits and character speeds for the terminal whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the terminal is quiescent. The input and output speeds are set from the first two bytes of the argument structure as indicated by the following table, which corresponds to the speeds supported by the DH-11 interface.

0	(hang up modem)
1	50 baud
2	75 baud
3	110 baud
4	134.5 baud
5	150 baud
6	200 baud
7	300 baud
8	600 baud
9	1200 baud
10	1800 baud
11	2400 baud
12	4800 baud
13	9600 baud
14	External A
15	External B

In the current configuration, only 110, 150 and 300 baud are really supported on dial-up lines. The half-duplex line discipline required for the 202 modem (1200 baud) is not supplied.

The next two characters of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The *mode* contains several bits that determine the system's treatment of the terminal:

100000	select one of two types of backspace delays
040000	select one of two types of form-feed and vertical-tab delays
030000	select one of four types of carriage-return delays
006000	select one of four types of tab delays
001400	select one of four types of new-line delays
000200	even parity allowed on input

000100 odd parity allowed on input
000040 raw mode
000020 map CR into LF; echo LF or CR as CR-LF
000010 echo (full duplex)
000004 map upper case to lower on input
000002 echo and print tabs as spaces
000001 hang up (drop 'data terminal ready') after last close

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently unimplemented.

Form-feed/vertical-tab delay type 1 lasts about 2 seconds.

Carriage-return delay types 1 and 2 last about .09 seconds, and type 3 lasts about .15 seconds. Types 2 and 3 have the side effect of not transmitting a carriage-return if at the leftmost column.

New-line delay type 1 is dependent on the current column and is tuned for the Model 37 TELETYPE®. Type 2 lasts about .03 seconds and type 3 lasts about .15 seconds.

Tab delay type 1 is dependent on the amount of movement and is tuned for the Model 37 TELETYPE. Other types are unimplemented and are 0.

Characters with the wrong parity, as determined by bits 0200 and 0100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file character (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially.

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (used for terminals without the newline function, i.e. most).

The upper case mode is used on terminals without lower case, see *tty(IV)*.

The hangup mode 01 causes the line to be disconnected when the last process with the line open closes it or terminates. It is useful when a port is to be used for some special purpose; for example, if it is associated with an ACU used to place outgoing calls.

This system call is also used with certain special files other than terminals, and is system dependent.

SEE ALSO

stty(I), gtty(II), tty(IV)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a terminal. From C, a negative value indicates an error.

NAME

sync – update super-block

SYNOPSIS

(sync = 36.)

sys sync

DESCRIPTION

Sync causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *icheck, df,* etc. It is mandatory before a boot.

SEE ALSO

sync(I)

NAME

tell – get file offset

SYNOPSIS

(tell = 40.)
(file descriptor in r0)
sys tell
(offset in r0-r1)

long tell(file)
int file;

DESCRIPTION

Tell returns the current read/write pointer associated with the open file whose descriptor is specified as argument.

SEE ALSO

seek (II)

DIAGNOSTICS

C-bit set or −1 returned for an unknown file descriptor.

NAME

time – get date and time

SYNOPSIS

(time = 13.)
sys time

time(tvec)
long *tvec;

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. From *as*, the high order word is in the r0 register and the low order is in r1. From C, the user-supplied vector is filled in.

SEE ALSO

date(I), stime(II), ctime(III)

NAME

times – get process times

SYNOPSIS

(times = 43.)
sys times; buffer

times(buffer)
struct tbuffer *buffer;

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/60 seconds.

After the call, the buffer will appear as follows:

```
struct tbuffer {
        long    proc_user_time;
        long    proc_system_time;
        long    child_user_time;
        long    child_system_time;
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time(I)

NAME

udata – get per-user data

SYNOPSIS

(pwbsys = 57.; udata = 1)
(pointer to buffer in r0)
(function in r1)
sys pwbsys; udata

DESCRIPTION

Udata is used to access a 32 byte section of the per-user process data region. If the *function* is zero, the section is read into the buffer given by the pointer. If the function is non-zero, the section is written from the buffer if super-user. The structure of the section is left to the user.

SEE ALSO

loginfo(I), loginfo(II)

DIAGNOSTICS

The error bit(c-bit) is set if the buffer can not be read or written, or if not super-user for write. From C, a −1 return indicates an error.

NAME

umount – dismount file system

SYNOPSIS

(umount = 22.)

sys umount; special

DESCRIPTION

Umount announces to the system that special file *special* is no longer to contain a removable file system. The file associated with the special file reverts to its ordinary interpretation; see *mount(II)*.

Only the super-user can execute umount.

SEE ALSO

umount(VIII), mount(II)

DIAGNOSTICS

Error bit (c-bit) set if no file system was mounted on the special file or if there are still active files on the mounted file system.

NAME

uname – get name of current PWB/UNIX

SYNOPSIS

(pwbsys = 57.; uname = 0)
(pointer to name in r0)
sys pwbsys; uname

uname(name)
char *name;

DESCRIPTION

Uname returns in *name* the 8 byte character name of the current PWB/UNIX. The name is not null-terminated. By convention, the name is of the form pwb?date. For example, pwba0401 would indicate that this is PWB/UNIX System A and that its operating system was last modified on April 1.

This function is kept in the **−lPW** library.

SEE ALSO

uname(I)

DIAGNOSTICS

The error bit(c-bit) is set if *name* can not be written. From C, a −1 return indicates an error.

NAME

unlink – remove directory entry

SYNOPSIS

(unlink = 10.)
sys **unlink; name**

unlink(name)
char *name;

DESCRIPTION

Name points to a null-terminated string. *Unlink* removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO

rm(I), rmdir(I), link(II)

DIAGNOSTICS

The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user). From C, a −1 return indicates an error.

NAME

ustat – get file system statistics

SYNOPSIS

```
(pwbsys = 57.; ustat = 2)
(pointer to buf in r0)
(device number in r1)
sys     pwbsys; ustat

ustat(device, buf)
char    *buf;
```

DESCRIPTION

Ustat is designed to return a section of the super block of the mounted file system specified by *device*. *Device* is *addr[0]* of the inode of the mounted block-type special file. The structure of *buf* is:

```
struct {
        int     s_tfree;        /* total free */
        int     s_tinode;       /* total inodes free */
        char    s_fname[6];     /* filsys name */
        char    s_fpack[6];     /* filsys pack name */
}
```

This function is kept in the **−lPW** library.

SEE ALSO

fs(V)

DIAGNOSTICS

The error bit(c-bit) is set if *device* is not mounted or *buf* can not be written. From C, a −1 return indicates an error.

NAME

utime – update times in file

SYNOPSIS

```
(pwbsys = 57.; utime = 3)
(pointer to times in r0)
(pointer to name in r1)
sys     pwbsys; utime

utime(name, times)
char *name, *times;
```

DESCRIPTION

Utime is used to set both the access and modification times of a file. Only the super-user may use this call. *Name* points to a null-terminated string naming a file, and *times* points to a structure containing two long integer time values:

```
struct {
        long int actime; /* access time */
        long int modtime;       /* modification time */
};
```

This function is kept in the **−lPW** library.

SEE ALSO

stat(II)

DIAGNOSTICS

The error bit(c-bit) is set if *name* does not exist, if not super-user, or if a read-only file system. From C, a −1 return indicates an error.

NAME

wait – wait for process to terminate

SYNOPSIS

(wait = 7.)
sys wait
(process ID in r0)
(status in r1)

wait(status)
int *status;

DESCRIPTION

Wait causes its caller to delay until one of its child processes terminates. If any child has died since the last *wait*, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of −1 returned). The normal return yields the process ID of the terminated child (in r0). In the case of several children several *wait* calls are needed to learn of all the deaths.

If no error is indicated on return, the r1 high byte (resp. the high byte stored into *status*) contains the low byte of the child process r0 (resp. the argument of *exit*) when it terminated. The r1 (resp. *status*) low byte contains the termination status of the process. See *signal*(II) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See *ptrace*(II). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit(II), fork(II), signal(II)

DIAGNOSTICS

The error bit (c-bit) is set if there are no children not previously waited for. From C, a returned value of −1 indicates an error.

NAME

write – write on a file

SYNOPSIS

(write = 4.)
(file descriptor in r0)
sys **write; buffer; nbytes**

write(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call.

Buffer is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned (in r0). It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat(II), open(II), pipe(II)

DIAGNOSTICS

The error bit (c-bit) is set on an error: bad descriptor, buffer address, or count; physical I/O errors. From C, a returned value of −1 indicates an error.

NAME

abort – generate an IOT fault

SYNOPSIS

abort()

DESCRIPTION

Abort executes the IOT instruction. This is usually considered a program fault by the system and results in termination with a core dump. It is used to generate a core image for debugging.

SEE ALSO

db(I), cdb(I), signal(II)

DIAGNOSTICS

usually "IOT trap -- core dumped" from the Shell.

NAME

abs, fabs — absolute value

SYNOPSIS

```
abs(i)
int i;

double fabs(x)
double x;
```

NAME

alloc, free – core allocator

SYNOPSIS

char *alloc(size)

free(ptr)
char *ptr;

DESCRIPTION

Alloc and *free* provide a simple general-purpose core management package. *Alloc* is given a size in bytes; it returns a pointer to an area at least that size which is even and hence can hold an object of any type. The argument to *free* is a pointer to an area previously allocated by *alloc*; this space is made available for further allocation.

Needless to say, grave disorder will result if the space assigned by *alloc* is overrun or if some random number is handed to *free*.

The routine uses a first-fit algorithm which coalesces blocks being freed with other blocks already free. It calls *sbrk* (see *break(II)*) to get more core from the system when there is no suitable space already free.

DIAGNOSTICS

Returns −1 if there is no available core.

BUGS

Allocated memory contains garbage instead of being cleared.

NAME

atan, atan2 – arc tangent function

SYNOPSIS

jsr pc,atan[2]

double atan(x)
double x;

double atan2(x, y)
double x, y;

DESCRIPTION

The *atan* entry returns the arc tangent of fr0 in fr0, from C the arc tangent of x is returned. The range is $-\pi/2$ to $\pi/2$. The *atan2* entry returns the arc tangent of fr0/fr1 in fr0, from C, the arc tangent of *x/y* is returned. The range is $-\pi$ to π.

DIAGNOSTIC

There is no error return.

NAME

atof – convert ASCII to floating

SYNOPSIS

double atof(nptr)
char *nptr;

DESCRIPTION

Atof converts a string to a floating number. *Nptr* should point to a string containing the number; the first unrecognized character ends the number.

The only numbers recognized are: an optional minus sign followed by a string of digits optionally containing one decimal point, then followed optionally by the letter e followed by a signed integer.

DIAGNOSTICS

There are none; overflow results in a very large number and garbage characters terminate the scan.

BUGS

The routine should accept initial +, initial blanks, and E for e. Overflow should be signaled.

NAME

atoi – convert ASCII to integer

SYNOPSIS

atoi(nptr)
char *nptr;

DESCRIPTION

Atoi converts the string pointed to by *nptr* to an integer. The string can contain leading blanks or tabs, an optional '-', and then an unbroken string of digits. Conversion stops at the first non digit.

SEE ALSO

atof(III)

BUGS

There is no provision for overflow.

NAME

cgetpid – return character form of process ID

SYNOPSIS

cgetpid(sptr) char *sptr;

DESCRIPTION

The *cgetpid* function appends the current UNIX process number to the string passed by the user. The character value is zero padded on the left to five digits.

The passed string is scanned left-to-right for the first *NUL* byte. If the process number were "123" and the function called as

```
s = "abc\0xxxxx";
cgetpid( s );
```

the value returned would be

"abc00123\0".

This function is kept in the **−lPW** library.

NAME

crypt – password encoding

SYNOPSIS

```
mov     $key,r0
jsr     pc,crypt

char *crypt(key)
char *key;
```

DESCRIPTION

On entry, r0 points to a string of characters terminated by an ASCII NUL. The routine performs an operation on the key which is difficult to invert (i.e. encrypts it) and leaves the resulting eleven bytes of ASCII alphanumerics in a static location.

From C, the *key* argument is a string and the value returned is a pointer to the eleven-character result.

This routine is used to encrypt all passwords.

SEE ALSO

passwd(I), passwd(V), login(I)

BUGS

Short or otherwise simple passwords can be decrypted easily by exhaustive search. Six characters of gibberish is reasonably safe.

NAME

ctime, localtime, gmtime – convert date and time to ASCII

SYNOPSIS

```
char *ctime(tvec)
int tvec[2];

[from Fortran]
double precision ctime
... = ctime(dummy)

int *localtime(tvec)
int tvec[2];

int *gmtime(tvec)
int tvec[2];
```

DESCRIPTION

Ctime converts a time in the vector *tvec* such as returned by *time*(II) into ASCII and returns a pointer to a character string in the form

Sun Sep 16 01:03:52 1973\n\0

All the fields have constant width.

The *localtime* and *gmtime* entries return pointers to integer vectors containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. The value is a pointer to an array whose components are

0 seconds
1 minutes
2 hours
3 day of the month (1-31)
4 month (0-11)
5 year – 1900
6 day of the week (Sunday = 0)
7 day of the year (0-365)
8 Daylight Saving Time flag if non-zero

The external variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, is 5*60*60); the external variable *daylight* is non-zero if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

A routine named *ctime* is also available from Fortran. Actually it more resembles the *time*(II) system entry in that it returns the number of seconds since the epoch 0000 GMT Jan. 1, 1970 (as a floating-point number).

SEE ALSO

time(II), regen(VIII)

NAME

descend – search UNIX file system directories

SYNOPSIS

```
int descend(name, goal, function, arg)
char *name, goal;
int (*function)();
??? arg;
```

DESCRIPTION

The *descend* function requires a file or directory name as first argument. If *name* is a directory name, *descend* recurses until regular files are found. Depending on the *goal* argument, the user-passed function *function* is called as follows:

```
(*function)(arg, name)
```

In addition to these arguments, *stat(II)* information is available for the current file. The external file status buffer is named "_Dstatb".

The *goal* argument is defined as:

'f'	call user function when *name* is a **file.**
'd'	call user function when *name* is a **directory.**
'b'	call user function for both.

This function is kept in the **−lPW** library.

DIAGNOSTICS

Descend returns zero on failure. It also writes error messages on file descriptor 2 (such as "——unreadable" for private files).

NAME

ecvt, fcvt — output conversion

SYNOPSIS

jsr pc,ecvt

jsr pc,fcvt

char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
...

DESCRIPTION

Ecvt is called with a floating point number in fr0.

On exit, the number has been converted into a string of ascii digits in a buffer pointed to by r0. The number of digits produced is controlled by a global variable *_ndigits*.

Moreover, the position of the decimal point is contained in r2: r2=0 means the d.p. is at the left hand end of the string of digits; r2>0 means the d.p. is within or to the right of the string.

The sign of the number is indicated by r1 (0 for +; 1 for −).

The low order digit has suffered decimal rounding (i. e. may have been carried into).

From C, the *value* is converted and a pointer to a null-terminated string of *ndigit* digits is returned. The position of the decimal point is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

Fcvt is identical to *ecvt*, except that the correct digit been rounded for F-style output of the number of digits specified by *ndigits*.

SEE ALSO

printf(III)

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. Instead, their addresses coincide with the first address above the program text region *(etext)*, above the initialized data region *(edata)*, or uninitialized data region *(end)*. The last is the same as the program break. Values are given to these symbols by the link editor *ld(I)* when, and only when, they are referred to but not defined in the set of programs loaded.

The usage of these symbols is rather specialized, but one plausible possibility is

```
extern end;
...
... = brk(&end+...);
```

The problem with this is that it ignores any other subroutines which may want to extend core for their purposes; these include *sbrk*, *alloc(III)*, and also secret subroutines invoked by the profile (–p) option of *cc*. Of course it was for the benefit of such systems that the symbols were invented, and user programs, unless they are in firm control of their environment, are wise not to refer to the absolute symbols directly.

One technique sometimes useful is to call *sbrk(0)*, which returns the value of the current program break, instead of referring to *&end*, which yields the program break at the instant execution started.

These symbols are accessible from assembly language if it is remembered that they should be prefixed by "_".

SEE ALSO

break(II), alloc(III)

NAME

exp – exponential function

SYNOPSIS

jsr pc,exp

double exp(x)
double x;

DESCRIPTION

The exponential of fr0 is returned in fr0. From C, the exponential of *x* is returned.

DIAGNOSTICS

If the result is not representable, the c-bit is set and the largest positive number is returned. From C, no diagnostic is available.

Zero is returned if the result would underflow.

NAME

floor, ceil – floor and ceiling functions

SYNOPSIS

double floor(x)
double x;

double ceil(x)
double x;

DESCRIPTION

The floor function returns the largest integer (as a double precision number) not greater than x.

The ceil function returns the smallest integer not less than x.

NAME

fmod – floating modulo function

SYNOPSIS

double fmod(x, y)
double x, y;

DESCRIPTION

Fmod returns the number f such that $x = iy + f$, i is an integer, and $0 \leqslant f < y$.

NAME

fptrap – floating point interpreter

SYNOPSIS

sys signal; 4; fptrap

DESCRIPTION

Fptrap is a simulator of the 11/45 FP11-B floating point unit. It works by intercepting illegal instruction traps and decoding and executing the floating point operation codes.

FILES

In systems with real floating point, there is a fake routine in /lib/liba.a with this name; when simulation is desired, the real version should be put in liba.a.

DIAGNOSTICS

A break point trap is given when a real illegal instruction trap occurs.

SEE ALSO

signal(II), the '–f' option of cc(I)

BUGS

Rounding mode is not interpreted. It's slow.

NAME

gamma – log gamma function

SYNOPSIS

jsr pc,gamma

double gamma(x);
double x;

DESCRIPTION

If x is passed (in fr0) *gamma* returns $\ln|\Gamma(x)|$ (in fr0). The sign of $\Gamma(x)$ is returned in the external integer *signgam*. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.)
        error( );
y = exp(y);
if(signgam)
        y = -y;
```

DIAGNOSTICS

The c-bit is set on negative integral arguments and the maximum value is returned. There is no error return for C programs.

BUGS

No error return from C.

NAME

getarg, iargc – get command arguments from Fortran

SYNOPSIS

call getarg (i, iarray [, isize])

... = iargc(dummy)

DESCRIPTION

The *getarg* entry fills in *iarray* (which is considered to be *integer)* with the Hollerith string representing the *i* th argument to the command in which it it is called. If no *isize* argument is specified, at least one blank is placed after the argument, and the last word affected is blank padded. The user should make sure that the array is big enough.

If the *isize* argument is given, the argument will be followed by blanks to fill up *isize* words, but even if the argument is long no more than that many words will be filled in.

The blank-padded array is suitable for use as an argument to *setfil(III).*

The *iargc* entry returns the number of arguments to the command, counting the first (file-name) argument.

SEE ALSO

exec(II), setfil(III)

NAME

getc, getw, fopen — buffered input

SYNOPSIS

```
mov     $filename,r0
jsr     r5,fopen; iobuf

fopen(filename, iobuf)
char *filename;
struct buf *iobuf;

jsr     r5,getc; iobuf
(character in r0)

getc(iobuf)
struct buf *iobuf;

jsr     r5,getw; iobuf
(word in r0)

getw(iobuf)
struct buf *iobuf;
```

DESCRIPTION

These routines provide a buffered input facility. *Iobuf* is the address of a 518(10) byte buffer area whose contents are maintained by these routines. Its structure is

```
struct buf {
        int fildes;     /* File descriptor */
        int nleft;      /* Chars left in buffer */
        char *nextp;    /* Ptr to next character */
        char buff[512]; /* The buffer */
};
```

Fopen may be called initially to open the file. On return, the error bit (c-bit) is set if the open failed. If *fopen* is never called, *get* will read from the standard input file. From C, the value is negative if the open failed.

Getc returns the next byte from the file in r0. The error bit is set on end of file or a read error. From C, the character is returned as an integer, without sign extension; it is −1 on end-of-file or error.

Getw returns the next word in r0. *Getc* and *getw* may be used alternately; there are no odd/even problems. *Getw* is may be called from C; −1 is returned on end-of-file or error, but of course is also a legitimate value.

Iobuf must be provided by the user; it must be on a word boundary.

To reuse the same buffer for another file, it is sufficient to close the original file and call *fopen* again.

Use the new "Standard I/O" instead.

SEE ALSO

open(II), read(II), getchar(III), putc(III)
A New Input-Output Package by D. M. Ritchie.

DIAGNOSTICS

c-bit set on EOF or error; from C, negative return indicates error or EOF. Moreover, *errno* is set by this routine just as it is for a system call (see *intro*(II)).

NAME

getchar – read character

SYNOPSIS

getchar( )

DESCRIPTION

Getchar provides the simplest means of reading characters from the standard input for C programs. It returns successive characters until end-of-file, when it returns "\0".

Associated with this routine is an external variable called *fin*, which is a structure containing a buffer such as described under *getc(III)*.

Generally speaking, *getchar* should be used only for the simplest applications; *getc* is better when there are multiple input files.

Use the new "Standard I/O" instead.

SEE ALSO

getc(III)
A New Input-Output Package by D. M. Ritchie.

DIAGNOSTICS

Null character returned on EOF or error.

BUGS

–1 should be returned on EOF; null is a legitimate character.

NAME

getpw – get name from UID

SYNOPSIS

getpw(uid, buf)
char *buf;

DESCRIPTION

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

passwd(V)

DIAGNOSTICS

non-zero return on error.

NAME

gmatch – match a string with a pattern (like glob(VIII))

SYNOPSIS

gmatch(string,pattern)
char *string, *pattern;

DESCRIPTION

Gmatch acts just like (is copied from) the *glob* command. It returns zero on failure and one on success. The characters '?', '[' and '*' have the usual meanings.

This function is kept in the **−lPW** library.

SEE ALSO

sh(I), glob(VIII)

NAME

hmul – high-order product

SYNOPSIS

hmul(x, y)

DESCRIPTION

Hmul returns the high-order 16 bits of the product of **x** and **y**. (The binary multiplication operator generates the low-order 16 bits of a product.)

NAME

ierror – catch Fortran errors

SYNOPSIS

if (ierror (errno) .ne. 0) goto label

DESCRIPTION

Ierror provides a way of detecting errors during the running of a Fortran program. Its argument is a run-time error number such as enumerated in *fc(I)*.

When *ierror* is called, it returns a 0 value; thus the **goto** statement in the synopsis is not executed. However, the routine stores inside itself the call point and invocation level. If and when the indicated error occurs, a **return** is simulated from *ierror* with a non-zero value; thus the **goto** (or other statement) is executed. It is a ghastly error to call *ierror* from a subroutine which has already returned when the error occurs.

This routine is essentially tailored to catching end-of-file situations. Typically it is called just before the start of the loop which reads the input file, and the **goto** jumps to a graceful termination of the program.

There is a limit of 5 on the number of different error numbers which can be caught.

SEE ALSO

fc(I)

BUGS

There is no way to ignore errors.

NAME

ldiv, lrem – long division

SYNOPSIS

ldiv (hidividend, lodividend, divisor)

lrem (hidividend, lodividend, divisor)

DESCRIPTION

The concatenation of the signed 16-bit *hidividend* and the unsigned 16-bit *lodividend* is divided by *divisor*. The 16-bit signed quotient is returned by *ldiv* and the 16-bit signed remainder is returned by *lrem*. Divide check and erroneous results will occur unless the magnitude of the divisor is greater than that of the high-order dividend.

An integer division of an unsigned dividend by a signed divisor may be accomplished by

```
quo = ldiv(0, dividend, divisor);
```

and similarly for the remainder operation.

Often both the quotient and the remainder are wanted. Therefore *ldiv* leaves a remainder in the external cell *ldivr*.

BUGS

No divide check check.

NAME

locv – long output conversion

SYNOPSIS

char *locv(hi, lo)
int hi, lo;

DESCRIPTION

Locv converts a signed double-precision integer, whose parts are passed as arguments, to the equivalent ASCII character string and returns a pointer to that string.

BUGS

Since *locv* returns a pointer to a static buffer containing the converted result, it cannot be used twice in the same expression; the second result overwrites the first.

NAME

log – natural logarithm

SYNOPSIS

```
jsr     pc,log

double log(x)
double x;
```

DESCRIPTION

The natural logarithm of fr0 is returned in fr0. From C, the natural logarithm of x is returned.

DIAGNOSTICS

The error bit (c-bit) is set if the input argument is less than or equal to zero and the result is a negative number very large in magnitude. From C, there is no error indication.

NAME

monitor – prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize)
int lowpc( ), highpc( ), buffer[ ], bufsize;

DESCRIPTION

Monitor is an interface to the *profil*(II) system call. *Lowpc* and *highpc* are the names of two functions; *buffer* is the address of a (user supplied) array of *bufsize* integers. *Monitor* arranges for the system to sample the user's program counter periodically and record the execution histogram in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
...
monitor(2, &etext, buf, bufsize);
```

Etext is a loader-defined symbol which lies just above all the program text.

To stop execution monitoring and write the results on the file *mon.out*, use

```
monitor(0);
```

Then, when the program exits, *prof*(I) can be used to examine the results.

It is seldom necessary to call this routine directly; the **-p** option of *cc* is simpler if one is satisfied with its default profile range and resolution.

FILES

mon.out

SEE ALSO

prof(I), profil(II), cc(I)

NAME

nargs – argument count

SYNOPSIS

nargs( )

DESCRIPTION

Nargs returns the number of actual parameters supplied by the caller of the routine which calls *nargs.*

The argument count is accurate only when none of the actual parameters is *float* or *double.* Such parameters count as four arguments instead of one.

BUGS

As indicated. Also, this routine does not work (and cannot be made to work) in programs with separated I and D space. Altogether it is best to avoid using this routine and depend, for example, on passing an explicit argument count.

NAME

nlist – get entries from name list

SYNOPSIS

```
nlist(filename, nl)
char *filename;
struct {
        char    name[8];
        int     type;
        int     value;
} nll ];
```

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of a list of 8-character names (null padded) each followed by two words. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are placed in the two words following the name. If the name is not found, the type entry is set to −1.

This subroutine is useful for examining the system name list kept in the file **/unix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(V)

DIAGNOSTICS

All type entries are set to −1 if the file cannot be found or if it is not a valid namelist.

NAME

perror, sys_errlist, sys_nerr, errno – system messages

SYNOPSIS

perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];

int errno;

DESCRIPTION

Perror produces a short error message describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno,* which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(II)

NAME

pexec – path search and execute a file

SYNOPSIS

```
char pathstr[128];
char shellnam[16];
pexec(name, argv)
char *name, *argv[];
pexinit( )
```

DESCRIPTION

Pexec provides an interface to the *execv* function that duplicates the shell's actions in searching for an executable file in a list of directories, as specified in the user's '.path' file.

Pexinit investigates the external arrays *pathstr* and *shellnam*. If either array is non-null, it leaves that array alone. If *pathstr* is null, it attempts to open the user's '.path' file and place the first line found there into *pathstr*, to be used later as a list of directories to be searched. If '.path' cannot be opened, it uses:

```
/bin:/etc:/      for super-user
:/bin:/usr/bin   for anyone else
```

If a second line is found in the '.path' file, it is taken as the name of the shell to be executed to interpret a shell procedure. If none is found, '/bin/sh' is used. *Pexinit* returns 0 to show successful completion, guaranteeing both arrays filled, and −1 otherwise.

Pexec first calls *pexinit*, then searches for the named file and executes it. The existence of two functions permits *pexinit* to be called once, followed by many *fork/pexec* pairs.

This function is kept in the **−lPW** library.

SEE ALSO

sh(I), exec(II), fork(II)

DIAGNOSTICS

Items in parentheses refer to error names in *intro*(II).

"cannot read .path"
".path too long" (more than 128+16 = 144 bytes long)
"No shell!" (real trouble, cannot execute shell)
"too large" (ENOMEM)
"arg list too long" (E2BIG)
"file not executable" (EACCES, no x bits set in file mode)
"not found" (name could not be found at all)
"text busy" (ETXTBSY, should be very rare)

BUGS

A pathname generated by the search mechanism may not exceed 47 characters in length.

NAME

plot: openpl et al. — graphics interface

SYNOPSIS

```
openpl( )

erase( )

label(s)
char s[ ];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

dot(x, y, dx, n, pattern)
int pattern[ ];

move(x, y)

cont(x, y)

point(x, y)

linemod(s)
char s[ ];

space(x0, y0, x1, y1)

closepl( )
```

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot(V)* for a description of their effect. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

String arguments to *label* and *linemod* are null-terminated, and do not contain newlines.

There are five libraries containing these routines, one that produces general graphics commands on the standard output, and one each for the vt0 storage scope, the GSI 300 terminal, the GSI 300S terminal, the DASI 450 terminal and the Tektronix 4014 terminal.

FILES

/sys/source/plot	directory containing the libraries above

SEE ALSO

graph(I), plot(I), plot(V)

NAME

pow – floating exponentiation

SYNOPSIS

```
movf    x,fr0
movf    y,fr1
jsr     pc,pow

double pow(x,y)
double x, y;
```

DESCRIPTION

Pow returns the value of x^y (in fr0). *Pow(0.0, y)* is 0 for any *y*. *Pow(−x, y)* returns a result only if *y* is an integer.

SEE ALSO

exp(III), log(III).

DIAGNOSTICS

The carry bit is set on return in case of overflow, *pow(0.0, 0.0)*, or *pow(−x, y)* for non-integral *y*. From C there is no diagnostic.

NAME

printf – formatted print

SYNOPSIS

printf(format, arg$_1$, ...);
char *format;

DESCRIPTION

Printf converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf.*

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign "–" which specifies *left adjustment* of the converted argument in the indicated field;
- an optional digit string specifying a *field width;* if the converted argument has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width;
- an optional period "." which serves to separate the field width from the next digit string;
- an optional digit string *(precision)* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are

d
o
x The integer argument is converted to decimal, octal, or hexadecimal notation respectively.

u The argument is taken to be an unsigned integer which is converted to decimal and printed (the result will be in the range 0 to 65535).

D
O
X The long integer argument is converted to decimal, octal, or hexadecimal notation respectively.

U The argument is taken to be an unsigned long integer which is converted to decimal and printed (the result will be in the range 0 to 4294967295).

f The argument is converted to decimal notation in the style "[–]ddd.ddd" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed. The argument should be *float* or *double.*

e The argument is converted in the style "[–]d.ddde±dd" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The argument should be a *float* or *double* quantity.

c The argument character is printed.

s The argument is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

r The argument is taken to be the address of a *printf* argument list (i.e., a vector of *printf* arguments). The current argument list is discarded, and the new list is used.

The "r" format can be used in the following situation:
"error()" is a subroutine which takes *printf* arguments (e.g., error("can't open %s", file);).
The source code for error() is:

```
error(arglist)
{
        printf("%r", &arglist);
        exit(1);
}
```

If no recognizable character appears after the %, that character is printed; thus % may be printed by use of the string %%. In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by calling *putchar*.

SEE ALSO

putchar (III)

BUGS

Very wide fields (>128 characters) fail.

NAME

putc, putw, fcreat, fflush – buffered output

SYNOPSIS

```
mov     $filename,r0
jsr     r5,fcreat; iobuf

fcreat(file, iobuf)
char *file;
struct buf *iobuf;

(get byte in r0)
jsr     r5,putc; iobuf

putc(c, iobuf)
int c;
struct buf *iobuf;

(get word in r0)
jsr     r5,putw; iobuf

putw(w, iobuf);
int w;
struct buf *iobuf;

jsr     r5,flush; iobuf

fflush(iobuf)
struct buf *iobuf;
```

DESCRIPTION

Fcreat creates the given file (mode 666) and sets up the buffer *iobuf* (size 518 bytes); *putc* and *putw* write a byte or word respectively onto the file; *flush* forces the contents of the buffer to be written, but does not close the file. The structure of the buffer is:

```
struct buf {
        int fildes;     /* File descriptor */
        int nunused;    /* Remaining slots */
        char *xfree;    /* Ptr to next free slot */
        char buff[512]; /* The buffer */
};
```

Before terminating, a program should call *flush* to force out the last of the output (*fflush* from C).

The user must supply *iobuf*, which should begin on a word boundary.

To write a new file using the same buffer, it suffices to call *[f]flush*, close the file, and call *fcreat* again.

Use the new "Standard I/O" instead.

SEE ALSO

creat(II), write(II), getc(III)
A New Input-Output Package by D. M. Ritchie.

DIAGNOSTICS

Fcreat sets the error bit (c-bit) if the file creation failed (from C, returns −1). *Putc* and *putw* return their character (word) argument. In all calls *errno* is set appropriately to 0 or to a system

error number. See intro(II).

NAME

putchar, flush – write character

SYNOPSIS

putchar(ch)

flush( )

DESCRIPTION

Putchar writes out its argument and returns it unchanged. Only the low-order byte is written, and only if it is non-null. Unless other arrangements have been made, *putchar* writes in unbuffered fashion on the standard output file.

Associated with this routine is an external variable *fout* which has the structure of a buffer discussed under *putc*(III). If the file descriptor part of this structure (first word) is greater than 2, output via *putchar* is buffered. To achieve buffered output one may say, for example,

```
fout = dup(1);          or
fout = creat(...);
```

In such a case *flush* must be called before the program terminates in order to flush out the buffered output. *Flush* may be called at any time.

Use the new "Standard I/O" instead.

SEE ALSO

putc(III)
A New Input-Output Package by D. M. Ritchie.

BUGS

The *fout* notion is kludgy.

NAME

qsort — quicker sort

SYNOPSIS

qsort(base, nel, width, compar)
char *base;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(I)

NAME

rand, srand – random number generator

SYNOPSIS

(seed in r0)

```
jsr     pc,srand        /to initialize
jsr     pc,rand /to get a random number
srand(seed)
int seed;
rand( )
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator to return successive pseudo-random numbers (in r0) in the range from 0 to $2^{15}-1$.

The generator is reinitialized by calling *srand* with 1 as argument (in r0). It can be set to a random starting point by calling *srand* with whatever you like as argument, for example the low-order word of the time.

BUGS

The low-order bits are not very random.

NAME

regcmp, regex – compile and execute regular expressions

SYNOPSIS

char *regcmp(string1[,string2,...],0);
char *string1, *string2, ...;

char *regex(re,subject[,ret0,...]);
char *re, *subject, *ret0, ...;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. The regular expression is the concatenation of *string1, string2,* etc. *Alloc(III)* is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A zero return from *regcmp* indicates an incorrect argument. *Regcmp(I)* has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern (*re*) against the *subject* string. Additional arguments are passed to receive values back. *Regex* returns zero on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(I); however, the syntax and semantics have been changed slightly.

symbols	*meaning*
[ ] * . ^	These symbols retain their current meaning.
$	Matches the end of the string; '\n' matches the newline.
–	Within brackets the minus means *through.* For example, [a–z] is equivalent to [abcd ... xyz]. The '–' can appear as itself only if used as the last or first character. For example, the character class expression [ ]–] matches the characters ']' and '–'.
+	A regular expression followed by '+' means *one or more times.* For example, [0–9]+ is equivalent to [0–9][0–9]*
{*m*} {*m*,} {*m*,*u*}	Integer values enclosed in { } indicate the number of times the preceeding regular expression is to be applied. *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present,e.g. {m}, *m* indicates the exact number of times the regular expression is to be applied. {m,} is analogous to {m,infinity}. The plus ('+') and star ('*') operations are equivalent to {1,} and {0,} respectively.
(...)$*n*	The value of the enclosed regular expression is to be returned. The matched string will be copied into the area pointed to by the *retn* argument (see the examples below). At present, at most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.
(...)	Parentheses are used for grouping. An operator, e.g. *,+,{}, can work on a single character or a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

Of necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

Example 1:

```
char *cursor, *newcursor, *ptr;
...
newcursor = regex((ptr=regcmp("^\n",0)),cursor);
free(ptr);
```

This example will match a leading newline in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0",0);
newcursor = regex(name,"123Testing321",ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (i.e., *newcursor* will point to the substring "21"). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
...
newcursor = regex(name,string);
```

This example applies a precompiled regular expression in *file.i* against *string* (see *regcmp*(I)).

Regcmp and *regex* are kept in the **−lPW** library.

SEE ALSO

regcmp(I), ed(I), alloc(III)

BUGS

The user program may run out of memory if regcmp() is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *alloc(III)* re-uses the same vector saving time and space.

```
/* user's program */
        ...
alloc(n) {
static int rebuf[256];
        return &rebuf;
}
free(ptr)
char *ptr;
{}
```

NAME

reset, setexit — execute non-local goto

SYNOPSIS

setexit( )

reset( )

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setexit saves its stack environment in a static place for later use by *reset.*

Reset restores the environment saved by the last call of *setexit.* It then returns in such a way that execution continues as if the call of *setexit* had just returned. All accessible data have values as of the time *reset* was called.

The routine that called *setexit* must still be active when *reset* is called.

SEE ALSO

signal(II), setjmp(III)

NAME

reset, setexit – execute non-local goto

SYNOPSIS

setexit( )

reset( )

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setexit saves its stack environment in a static place for later use by *reset.*

Reset restores the environment saved by the last call of *setexit.* It then returns in such a way that execution continues as if the call of *setexit* had just returned. All accessible data have values as of the time *reset* was called.

The routine that called *setexit* must still be active when *reset* is called.

SEE ALSO

signal(II), setjmp(III)

NAME

setfil – specify Fortran file name

SYNOPSIS

call setfil (unit, hollerith-string)

DESCRIPTION

Setfil provides a primitive way to associate an integer I/O *unit* number with a file named by the *hollerith-string*. The end of the file name is indicated by a blank. Subsequent I/O on this unit number will refer to the file whose name is specified by the string.

Setfil should be called only before any I/O has been done on the *unit*, or just after doing a **rewind** or **endfile**. It is ineffective for unit numbers 5 and 6.

SEE ALSO

fc(I)

BUGS

The exclusion of units 5 and 6 is unwarranted.

NAME

setjmp, longjmp – execute non-local goto

SYNOPSIS

int save[3];

setjmp(save)

longjmp(save)

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *save* for later use by *longjmp*. It returns 0 on the initial call.

Longjmp restores the environment saved in *save* by *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned (with a nonzero value). All accessible automatic and register data have values as of the time *setjmp* was called.

The routine that called *setjmp* must still be active when *longjmp* is called.

Although these functions are similar in purpose to *setexit* and *reset*, they permit several *setjmp* calls to be made, each with a different *save*. *Longjmp* may then return to any of them by selecting the appropriate one.

SEE ALSO

signal(II), reset(III)

NAME

sin, cos – trigonometric functions

SYNOPSIS

jsr pc,sin (cos)

double sin(x)
double x;

double cos(x)
double x;

DESCRIPTION

The sine (cosine) of fr0 (resp. **x**), measured in radians, is returned (in fr0).

The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

NAME

sleep — suspend execution for interval

SYNOPSIS

sleep(seconds)

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. The routine is implemented by setting an alarm clock signal and pausing until it occurs. Thus any other use of this signal may be counterproductive.

SEE ALSO

alarm(II), pause(II)

NAME

sqrt – square root function

SYNOPSIS

jsr pc,sqrt

double sqrt(x)
double x;

DESCRIPTION

The square root of fr0 (resp. **x**) is returned (in fr0).

DIAGNOSTICS

The c-bit is set on negative arguments and 0 is returned. There is no error return for C programs.

BUGS

No error return from C.

NAME

strcpy, strcat, strcmp, strlen – operations on ASCII strings

SYNOPSIS

```
strcpy(s1, s2)
char *s1, *s2;

strcat(s1, s2)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strlen(s)
char *s;
```

DESCRIPTION

strcpy

The null-terminated character string *s2* is copied to the location pointed to by *s1*. The space pointed to by *s1* must be large enough.

strcat

The end (null byte) of *s1* is found and *s2* is copied to *s1* starting there. The space pointed to by *s1* must be large enough.

strcmp

The character strings *s1* and *s2* are compared. The result is positive, zero, or negative, depending on whether *s1* is greater than, equal to, or less than *s2* (according to the ASCII collating sequence), respectively.

strlen

The number of bytes in *s* up to but not including a null byte is returned.

NAME

ttyn – return name of current terminal

SYNOPSIS

jsr pc,ttyn

ttyn(file)

DESCRIPTION

Ttyn hunts up the last character of the name of the terminal which is the standard input (from *as*) or is specified by the argument *file* descriptor (from C). If *n* is returned, the terminal name is then "/dev/tty*n*".

x is returned if the indicated file does not correspond to a terminal.

NAME

cat – phototypesetter interface

DESCRIPTION

Cat provides the interface to a Graphic Systems C/A/T phototypesetter. Bytes written on the file specify font, size, and other control information as well as the characters to be flashed. The coding will not be described here.

Only one process may have this file open at a time. It is write-only.

FILES

/dev/cat

SEE ALSO

troff(I)

NAME

dh – DH-11 communications multiplexer

DESCRIPTION

Each line attached to the DH-11 communications multiplexer behaves as described in *tty(IV)*. Input and output for each line may independently be set to run at any of 16 speeds; see *stty(II)* for the encoding.

FILES

/dev/tty?

SEE ALSO

tty(IV), stty(II)

NAME

dn – DN-11 ACU interface

DESCRIPTION

The *dn?* files are write-only. The permissible codes are:

0-9	dial 0-9
:	dial *
;	dial #
–	3 second delay for second dial tone
=	end-of-number

The entire telephone number must be presented in a single *write* system call.

It is recommended that an end-of-number code be given even though not all ACU's actually require it.

FILES

/dev/dn?

SEE ALSO

dp(IV)

NAME

dp – DP-11, DU-11 synchronous line interface

DESCRIPTION

The *dp0* file is a data-set interface. *Read* and *write* calls to dp0 are unlimited, but this works best when restricted to less than 512 bytes. Each write call is sent as a single record. Seven bits from each byte are written along with an eighth odd parity bit. The sync characters must be user supplied. Each read call returns characters received from a single record. Seven bits are returned unaltered; the eighth bit is set if the byte was not received in odd parity. A 10 second time out is set and a zero-byte record is returned if nothing is received in that time. An error is returned if data-set ready is not present.

FILES

/dev/dp0

SEE ALSO

dn(IV)

NAME

hp – RP04/RP05/RP06 moving-head disk

DESCRIPTION

The files *rp0* ... *rp7* refer to sections of the RP04/RP05/RP06 disk drive 0. The files *rp10* ... *rp17* refer to drive 1, etc. This is done since the size of of a full pack is over 100,000 blocks and internally the system is only capable of addressing 65536 blocks. Also since the disk is so large, this allows it to be broken up into more manageable pieces.

The origin and size of the sections on each drive are as follows:

section	start	length
0	0	11286
1	27	53504
2	155	53504
3	283	53504
4	27	65535
5	184	65535
6	341	29260
7	unassigned	

The start address is a cylinder address, with each cylinder containing 418 blocks. For the RP06 drives, this table in the system must be changed to allow full addressing. It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter.

The *rp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rrp* and end with a number which selects the same disk section as the corresponding *rp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rp*, /dev/rrp*

NAME

hs – RS03/RS04 fixed-head disk

DESCRIPTION

The files *rs0* ... *rs7* refer to RS03 disk drives 0 through 7. The files *rs10* ... *rs17* refer to RS04 disk drives 0 through 7. The RS03 drives are each 1024 blocks long and the RS04 drives are each 2048 blocks long.

The *rs* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw HS files begin with *rrs*. The same minor device considerations hold for the raw interface as for the normal interface.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rs*, /dev/rrs*

NAME

ht – TU16 magtape interface

DESCRIPTION

The files *mt0, ..., mt15* refer to the Digital Equipment Corporation TU16 magnetic tape control and transports. The files *mt0, ..., mt7* are 800bpi, and the files *mt8, ..., mt15* are 1600bpi. The files *mt0, ..., mt3, mt8, ..., mt11* are designated normal-rewind on close, and the files *mt4, ..., mt7, mt12, ..., mt15* are no-rewind on close. When opened for reading or writing, the tape is assumed to be positioned as desired. When a file is closed, a double end-of-file (double tape mark) is written if the file was opened for writing. If the file was normal-rewind, the tape is rewound. If it is no-rewind and the file was open for writing, the tape is positioned before the second EOF just written. If the file was no-rewind and opened read-only, the tape is positioned after the EOF following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF or the last write. The EOF is returned as a zero-length read. By judiciously choosing *mt* files, it is possible to read and write multi-file tapes.

A standard tape consists of several 512 byte records terminated by an EOF. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time (although very inadvisable).

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the "raw" interface is appropriate. The associated files are named *rmt0, ..., rmt15*. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned after the EOF, so that the next read will return the next record.

FILES

/dev/mt*, /dev/rmt*

SEE ALSO

tp(I)

BUGS

If any non-data error is encountered, it refuses to do anything more until closed. The driver is limited to four transports.

NAME

kl – KL-11 or DL-11 asynchronous interface

DESCRIPTION

The discussion of terminal I/O given in *tty(IV)* applies to these devices.

Since they run at a constant speed, attempts to change the speed via *stty*(II) are ignored.

The on-line console terminal is interfaced using a KL-11 or DL-11. By appropriate switch settings during a reboot, UNIX will come up as a single-user system with I/O on the console terminal.

FILES

/dev/tty8 console

SEE ALSO

tty(IV), init(VIII)

BUGS

Full modem control for the DL-11E is not implemented.

NAME

lp – line printer

DESCRIPTION

Lp provides the interface to any of the standard Digital Equipment Corporation line printers. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

An internal parameter within the driver determines whether or not the device is treated as having a 96- or 64-character set. In half-ASCII mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

{	(+
}	)+
`	'-
|	!+
~	^-

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. A sequence of newlines which extends over the end of a page is turned into a form feed. All lines are indented 8 characters. Lines longer than 80 characters are truncated. These numbers are parameters in the driver; another parameter allows indenting all output if it is unpleasantly near the left margin.

FILES

/dev/lp

BUGS

In half-ASCII mode, the indent and the maximum line length should be settable by a call analogous to *stty(II)*.

NAME

mem, kmem, null – core memory

DESCRIPTION

Mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system using the debugger.

A memory address is a 22-bit quantity used to set up memory management to address the full memory space. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present. Especially since reads and writes are a byte at a time.

The file *kmem* is the same as *mem* except that the kernel virtual data address space rather than physical memory is accessed. In particular, the I/O area of *kmem* is located beginning at 160000 (octal) rather than at 760000. The 1K region beginning at 140000 (octal) is the system's data for the current process.

The file *null* returns end-of-file on *read* and ignores *write*.

FILES

/dev/mem, /dev/kmem, /dev/null

NAME

rje – DQS-11B interface for remote job entry

DESCRIPTION

The **rje** interface defines a special file that looks like a concatenation of Binary Synchronous Communication (BSC) text blocks. This file may be both written to and read from, but not simultaneously. Data transfer with the two-point BSC discipline is strictly half-duplex.

The device can be opened by only one process at a time. It is expected that a process that successfully opens the DQS will spawn separate subprocesses to handle reading and writing. However, no distinction is made among the several processes that may have the DQS open. For example, reads within a message, even from a single block, may be executed by several processes in sequence. The overriding constraint is that a complete message must be read from or written to the DQS before any transfer of data in the opposite direction can begin. A process that tries to write while the DQS is reading, or vice versa, will be put to sleep until the transfer of the currently active message has been completed.

A complete message consists of one or more text blocks. A message being written to the DQS is terminated by a write of zero bytes, which causes an EOT to be transmitted. A message being read from the DQS is terminated by the reception of an EOT (which is not passed on to the reader, but is registered as a read of zero bytes). By convention, an EOT follows each block which ends in an ETX.

The length of a text block cannot exceed 512 bytes, including the line prefix and appendix. These two sequences, which must be present in blocks being written and will be passed on in blocks read, are constructed from the control bytes SOH, STX, ETB, ETX, DLE. The DQS itself will supply leading SYN bytes and trailing block check and pad bytes. The interface examines only the last byte of each text block received and so is unaware of the presence of headings or transparent text. The selection and interpretation of these features is the user's responsibility.

Line control functions, such as the alternating affirmative responses (ACK0 and ACK1), are automatically interspersed with text blocks as required by the line discipline. The interface handles the initial line bid and the EOT reset at the end of a transmission. A 3-second time-out is also respected. The interface will send TTD's and respond WACK's if its buffers are not serviced fast enough. When receiving, expiration of the time-out will cause the interface to abort the active message by sending EOT. When transmitting, the failure to send a block successfully after seven tries will cause the interface to terminate the active message prematurely. Such aborts cannot be appealed.

Reads on the DQS will return bytes from a single text block. If one read does not exhaust a text block, successive reads will return additional bytes from the same block. A returned count of zero indicates the end of a message. Until the remote station bids for the line, all reads will return zero bytes. The error bit will never be set by the interface itself. The DQS must be read to the end of a message before it will accept writes.

Writes to the DQS must consist of a single, entire text block. A write that specifies a count of zero bytes defines the end of a message. The count returned by a write call must be checked. A count of zero for the first write of a new message indicates that it was not possible to acquire the line. Otherwise, the DQS should return exactly the count specified in the write call. However, the error bit is set when a line error requires that the message be aborted. Notification of the error is not punctual, because data blocks are buffered for transmission. A write of zero bytes must be issued, or an error must occur, before the DQS will accept reads.

An open returns with the error bit set if the DQS is already open or not ready. The DQS should be opened in mode 2 to allow both reading and writing.

The DQS interface steals a number of buffers from UNIX (currently two) for the duration of each message. This number is specified at system generation time and may be tuned to influence overall system throughput.

FILES

/dev/rjei DQS11-B communicating with IBM 370

SEE ALSO

General Information—Binary Synchronous Communication, IBM Systems Reference Library #GA27-3004.

DQS11-A/B PDP-11 Communications Controller Option Description, Digital Equipment Corporation.

NAME

rp – RP-11/RP03 moving-head disk

DESCRIPTION

The files *rp0* ... *rp7* refer to sections of the RP03 disk drive 0. The files *rp10* ... *rp17* refer to drive 1, etc. This is done since the size of a full pack is 81200 blocks and internally the system is only capable of addressing 65536 blocks. Also since the disk is so large, this allows it to be broken up into more manageable pieces.

The origin and size of the sections on each drive are as follows:

section	start	length
0	0	7600
1	38	36200
2	219	36200
3	40	65535
4	22	36200
5	203	40600
6-7	unassigned	

The start address is a cylinder address, with each cylinder containing 200 blocks. It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter.

The *rp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rrp* and end with a number which selects the same disk section as the corresponding *rp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rp*, /dev/rrp*

NAME

tm – TM11/TU10 magtape interface

DESCRIPTION

The files *mt0, ..., mt7* refer to the Digital Equipment Corporation TM11/TU10 magnetic tape control and transports at 800bpi. The files *mt0, ..., mt3* are designated normal-rewind on close and the files *mt4, ..., mt7* are no-rewind on close. When opened for reading or writing, the tape is assumed to be positioned as desired. When a file is closed, a double end-of-file (double tape mark) is written if the file was opened for writing. If the file was normal-rewind, the tape is rewound. If it is no-rewind and the file was open for writing, the tape is positioned before the second EOF just written. If the file was no-rewind and opened read-only, the tape is positioned after the EOF following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF or the last write. The EOF is returned as a zero-length read. By judiciously choosing *mt* files, it is possible to read and write multi-file tapes.

A standard tape consists of several 512 byte records terminated by an EOF. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time (although very inadvisable).

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the "raw" interface is appropriate. The associated files are named *rmt0, ..., rmt7*. Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, up to the buffer size specified. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned after the EOF, so that the next read will return the next record.

FILES

/dev/mt?, /dev/rmt?

SEE ALSO

tp(I)

BUGS

If any non-data error is encountered, it refuses to do anything more until closed. The driver is limited to four transports.

NAME

tty – general terminal interface

DESCRIPTION

This section describes both a particular special file, and the general nature of the terminal interface.

The file */dev/tty* is, in each process, a synonym for the control terminal associated with that process. It is useful for programs or Shell sequences which wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs which demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

As for terminals in general: all of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of the interface; *kl(IV)* and *dh(IV)* describe peculiarities of the individual devices..

When a terminal file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by *init(VIII)* and become a user's input and output file. The very first terminal file open in a process becomes the *control terminal* for that process. The control terminal plays a special role in handling quit or interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters which have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

These special files have a number of modes which can be changed by use of the *stty(II)*. When first opened, the interface mode is 300 baud; either parity accepted; 10 bits/character (one stop bit); and newline action character. Modes that can be changed by *stty* include the interface speed (if the hardware permits); acceptance of even parity, odd parity, or both; a raw mode in which all characters may be read one at a time, and all 8-bits are sent on output; a carriage return (CR) mode in which CR is mapped into newline on input and either CR or line feed (LF) cause echoing of the sequence LF-CR; mapping of upper case letters into lower case; suppression of echoing; a variety of delays after function characters; and the printing of tabs as spaces. See *getty*(VIII) for the way that terminal speed and type are detected.

Normally, terminal input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character '#' erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. By default, the character '@' kills the entire line up to the point where it was typed, but not beyond an EOT. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that may have been done. Either '@' or '#' may be entered literally by preceding it by '\'; the erase or kill character remains, but the '\' disappears. These two characters may be changed to others.

When desired, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. In addition, the following escape sequences are generated on output and accepted on input:

for	use
`	\'
|	\!
~	\^
{	\(
}	\)

In raw mode, the program reading is awakened on each character. No erase or kill processing is done; and the EOT, quit and interrupt characters are not treated specially. The input parity bit is passed back to the reader. On output, all 8-bits are sent.

The ASCII EOT (control-D) character may be used to generate an end of file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication. The EOT is passed back unchanged in raw mode.

When the carrier signal from the data-set drops (usually because the user has hung up his terminal) a *hangup* signal is sent to all processes with the terminal as control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs which read a terminal and test for end-of-file on their input can terminate appropriately when hung up on.

Two characters have a special meaning when typed. The ASCII DEL character (sometimes called 'rubout') is not passed to a program but generates an *interrupt* signal which is sent to all processes associated with the control terminal. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See *signal*(II).

The ASCII FS character generates the *quit* signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. The EOT character is not transmitted (except in raw mode) to prevent terminals which respond to it from hanging up.

FILES

/dev/tty

SEE ALSO

kl(IV), dh(IV), getty(VIII), stty(I), stty(II), gtty(II), signal(II)

BUGS

Half-duplex terminals are not supported.

NAME

a.out – assembler and link editor output

DESCRIPTION

A.out is the output file of the assembler *as* and the link editor *ld*. Both programs make *a.out* executable if there were no errors and no unresolved external references.

This file has four sections: a header, the program and data text, a symbol table, and relocation bits (in that order). The last two may be empty if the program was loaded with the "–s" option of *ld* or if the symbols and relocation have been removed by *strip*.

The header always contains 8 words:

1 A magic number (407, 410, or 411(8))
2 The size of the program text segment
3 The size of the initialized portion of the data segment
4 The size of the uninitialized (bss) portion of the data segment
5 The size of the symbol table
6 The entry location (always 0 at present)
7 Unused
8 A flag indicating relocation bits have been suppressed

The sizes of each segment are in bytes but are even. The size of the header is not included in any of the other sizes.

When a file produced by the assembler or loader is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number (word 0) is 407, it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 410, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. If the magic number is 411, the text segment is again pure, write-protected, and shared, and moreover instruction and data space are separated; the text and data segment both begin at location 0. See the 11/45 handbook for restrictions which apply to this situation.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *break* system call.

The start of the text segment in the file is 20(8); the start of the data segment is $20+S_t$ (the size of the text) the start of the relocation information is $20+S_t+S_d$; the start of the symbol table is $20+2(S_t+S_d)$ if the relocation information is present, $20+S_t+S_d$ if not.

The symbol table consists of 6-word entries. The first four words contain the ASCII name of the symbol, null-padded. The next word is a flag indicating the type of symbol. The following values are possible:

00 undefined symbol
01 absolute symbol
02 text segment symbol
03 data segment symbol
37 file name symbol (produced by ld)
04 bss segment symbol
40 undefined external (.globl) symbol

41 absolute external symbol
42 text segment external symbol
43 data segment external symbol
44 bss segment external symbol

Values other than those given above may occur if the user has defined some of his own instructions.

The last word of a symbol table entry contains the value of the symbol.

If the symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation bits for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the "suppress relocation" flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

00 indicates the reference is absolute
02 indicates the reference is to the text segment
04 indicates the reference is to initialized data
06 indicates the reference is to bss (uninitialized data)
10 indicates the reference is to an undefined external symbol.

Bit 0 of the relocation word indicates if *on* that the reference is relative to the pc (e.g. "clr x"); if *off*, that the reference is to the actual symbol (e.g., "clr *$x").

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

SEE ALSO

as(I), ld(I), strip(I), nm(I)

NAME

ar – archive (library) file format

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 177545(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 26 bytes long:

```
struct archive {
        char    a_name[14];     /* file name, null padded on right */
        long    a_date; /* modification time of file */
        char    a_uid;  /* user ID of file owner */
        char    a_gid;  /* group ID of file owner */
        int     a_mode;         /* file mode */
        long    a_size; /* file size */
};
```

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

FILES

/usr/include/archive.h

SEE ALSO

ar(I), ld(I)

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq	006	ack	007	bel
010	bs	011	ht	012	nl	013	vt	014	np	015	cr	016	so	017	si
020	dle	021	dc1	022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs	036	rs	037	us
040	sp	041	!	042	"	043	#	044	$	045	%	046	&	047	'
050	(	051	)	052	*	053	+	054	,	055	–	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	`	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	|	175	}	176	~	177	del

FILES

found in /usr/pub

NAME

checklist – list of file systems processed by check

DESCRIPTION

Checklist resides in directory **/etc** and contains a list of at most 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *check(VIII)* command.

SEE ALSO

check(VIII)

NAME

core – format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See *signal(II)* for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called "core" and is written in the process's working directory (provided it can be; normal access controls apply).

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*. Currently for PWB/UNIX systems it is 768 bytes. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment i read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system. The important stuff not detailed therein is the locations of the registers. Here are their offsets. The parenthesized numbers for the floating registers are used if the floating-point hardware is in single precision mode, as indicated in the status register.

fpsr	0004	
fr0	0006	(0006)
fr1	0036	(0022)
fr2	0046	(0026)
fr3	0056	(0032)
fr4	0016	(0012)
fr5	0026	(0016)

The following registers are located relative to end of the first section.

r0	−6
r1	−12
r2	−30
r3	−26
r4	−24
r5	−22
sp	−14
pc	−4
ps	−2

In general the debuggers *db(I)* and *cdb(I)* are sufficient to deal with core images.

SEE ALSO

adb(I), cdb(I), db(I), signal(II)

NAME

cpio – format of cpio archive

SYNOPSIS

```
struct {
        int     h_magic,
                h_dev,
                h_ino,
                h_mode,
                h_uid,
                h_gid,
                h_nlink,
                h_majmin;
        long    h_mtime;
        int     h_namesize;
        long    h_filesize;
        char    h_name[h_namesize rounded to word];
        char    data[h_filesize rounded to word];
} archive;
```

DESCRIPTION

The *contents* of each file is recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707. The items *h_dev* through *h_mtime* have meanings explained in *stat*(II). The length of the null-terminated pathname *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name 'TRAILER!!!'. Special files, directories, and the trailer are recorded with h_filesize = 0.

SEE ALSO

cpio(I), stat(II)

BUGS

This format should be reconciled with *archive*(V).

NAME

directory – format of directories

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2-15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots. The structure is:

```
struct dir {
        int     d_ino;  /* i-number */
        char    d_name[14];     /* file name */
};
```

By convention, the first two entries in each directory are for "." and "..". The first is an entry for the directory itself. The second is for the parent directory. The meaning of ".." is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases ".." has the same meaning as ".".

FILES

/usr/include/dir.h

SEE ALSO

fs(V)

NAME

dump – incremental dump tape format

DESCRIPTION

The *dump*(VIII) and *restor*(VIII) commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of blocks of 512-bytes each. The first block has the following structure.

```
struct {
        int     isize;
        int     fsize;
        int     date[2];
        int     ddate[2];
        int     tsize;
};
```

Isize and *fsize* are the corresponding values from the super block of the dumped file system (see *fs*(V)). *Date* is the date of the dump. *Ddate* is the incremental dump date. The incremental dump contains all files modified between *ddate* and *date*. *Tsize* is the number of blocks per reel. This block checksums to the octal value 031415.

Next there are enough whole tape blocks to contain one word per file of the dumped file system. This is *isize* divided by 16 rounded to the next higher integer. The first word corresponds to i-node 1, the second to i-node 2, and so forth. If a word is zero, then the corresponding file exists, but was not dumped. (Was not modified after *ddate*.) If the word is −1, the file does not exist. Other values for the word indicate that the file was dumped and the value is one more than the number of blocks it contains.

The rest of the tape contains for each dumped file a header block and the data blocks from the file. The header contains an exact copy of the i-node (see *fs*(V)) and also checksums to 031415. The next-to-last word of the block contains the tape block number, to aid in (unimplemented) recovery after tape errors. The number of data blocks per file is directly specified by the control word for the file and indirectly specified by the size in the i-node. If these numbers differ, the file was dumped with a 'phase error'.

SEE ALSO

dump(VIII), restor(VIII), fs(V)

NAME

ebcdic – file format

DESCRIPTION

The ebcdic format is a convenient representation for files consisting of card images in an arbitrary code. Files created by the *send*(I) command, to be entered into rje *xmit** queues, are in this format. So are files of punch output from HASP.

An ebcdic file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

SEE ALSO

send(I), hasp(VIII)

NAME

fs – format of file system volume

DESCRIPTION

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a commo format for certain vital information. Every such volume is divided into a certain number of 25 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program, pac label, or other information.

Block 1 is the *super block.* Starting from its first word, the format of a super-block is

```
struct  {
        unsigned  int     isize;
        unsigned  int     fsize;
                  int     nfree;
        unsigned  int     free[100];
                  int     ninode;
        unsigned  int     inode[100];
                  char    flock;
                  char    ilock;
                  char    fmod;
                  char    ronly;
        long      int     time;
                  int     pad[40];
        unsigned  int     tfree;
        unsigned  int     tinode;
                  char    fname[6];
                  char    fpack[6];
};
```

Isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. *Fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *free* array contains, in *free[1]*, ... , *free[nfree−1]*, up to 99 numbers of free blocks. *Free[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement *nfree*, and the new block is *free[nfree]*. If the new block number is 0, there are no blocks left, so give an error. If *nfree* became 0, read in the block named by the new block number, replace *nfree* by its first word, and copy the block numbers in the next 100 words into the *free* array. To free a block, check if *nfree* is 100; if so, copy *nfree* and the *free* array into it, write it out, and set *nfree* to 0. In any event set *free[nfree]* to the freed block's number and increment *nfree*.

Tfree is the total free blocks available in the file system.

Ninode is the number of free i-numbers in the *inode* array. To allocate an i-node: if *ninode* is greater than 0, decrement it and return *inode[ninode]*. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *inode* array, then try again. To free an i-node, provided *ninode* is less than 100, place its number into *inode[ninode]* and increment *ninode*. If *ninode* is already 100, don't bother to enter the freed i-node into any table. This list of i-nodes is

only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

Tinode is the total free inodes available in the file system.

Flock and *ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

Ronly is a read-only flag to indicate write-protection.

Time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1, 1970 (GMT). During a reboot, the *time* of the super-block for the root file system is used to set the system's idea of the time.

Fname is the name of the file system and *fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 32 bytes long, so 16 of them fit into a block. Therefore, i-node *i* is located in block (*i* + 31) / 16, and begins 32*((*i* + 31) (mod 16)) bytes from its start. I-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node is as follows.

```
struct {
        int     flags;          /* +0: see below */
        char    nlinks;         /* +2: number of links to file */
        char    uid;            /* +3: user ID of owner */
        char    gid;            /* +4: group ID of owner */
        char    size0;          /* +5: high byte of 24-bit size */
        int     size1;          /* +6: low word of 24-bit size */
        int     addr[8];        /* +8: block numbers or device number */
        int     actime[2];      /* +24: time of last access */
        int     modtime[2];     /* +28: time of last modification */
};
```

The flags are as follows:

```
100000  i-node is allocated
060000  2-bit file type:
        000000  plain file
        040000  directory
        020000  character-type special file
        060000  block-type special file.
010000  large file
004000  set user-ID on execution
002000  set group-ID on execution
000400  read (owner)
000200  write (owner)
000100  execute (owner)
000070  read, write, execute (group)
000007  read, write, execute (others)
```

Special files are recognized by their flags and not by i-number. A block-type special file is basically one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files the high byte of the first address

ord specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number *n* of a file is accessed as follows. *N* is divided by 512 to find its logical block number (say *b*) in the file. If the file is small (flag 010000 is 0), then *b* must be less than 8, and the physical block number is *addr[b]*.

If the file is large, *b* is divided by 256 to yield *i*. If *i* is less than 8, then *addr[i]* is the physical block number of the indirect block. The remainder from the division yields the word in the indirect block which contains the number of the block for the sought-for byte.

For block *b* in a file to exist, it is not necessary that all blocks less than *b* exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

FILES

/usr/include/filsys.h
/usr/include/stat.h

SEE ALSO

icheck(VIII), dcheck(VIII)

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on UNIX with non-standard tabs, *i.e.*, tabs which are not set at the simple interval of eight columns. Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets '<:' and ':>'. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a '–' followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a '–' followed by the name of a 'canned' tab specification.

Standard tabs are specified by 't–8' or, equivalently, 't1,9,17,25,etc'. The canned tabs which are recognized are defined by the *tabs*(I) command – a,a2,c,c2,c3,f,p,s,u.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are 't–8' and 'm0'. If the **s** parameter is not specified, no size checking is performed.

If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file.

The following is an example of a line containing a format specification:

```
/* <:t5,10,15 s72:> */
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several *Programmer's Workbench* commands correctly interpret the format specification for a file. Among them is *gath* which may be used to convert files to a standard format acceptable to other UNIX commands.

SEE ALSO

ed(I), gath(I), reform(I), send(I), tabs(I)

NAME

greek – graphics for extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek

DESCRIPTION

Greek gives the mapping from ascii to the "shift out" graphics in effect between SO and SI on a Model 37 TELETYPE® with a 128-character type-box. It contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ϵ	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	$\neg$	_
partial	∂	]	integral	$\int$	^			

SEE ALSO

ascii(V)

NAME

group – group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(I), login(I), crypt(III), passwd(I)

NAME

master – master device information table

DESCRIPTION

This file is used by the *config*(VIII) program to obtain device configuration information that enables it to generate the *low.s* and *conf.c* files.

The file consists of two parts, separated by a line with a dollar sign ($) in column 1. Part one contains device information, while part two contains names of devices that have aliases. Any line with an asterisk (*) in column 1 is treated as a comment.

Part one contains lines consisting of at least 10 fields and at most 13 fields, with the fields delimited by tabs and/or blanks, as follows:

Field 1: device name (8 characters maximum).

Field 2: interrupt vector size (decimal, in bytes).

Field 3: device mask – each "on" bit indicates that the handler exists, as follows:

000020	open handler
000010	close handler
000004	read handler
000002	write handler
000001	sgtty handler.

Field 4: device type indicator, as follows:

000020	immediate allocation
000010	block device
000004	character device
000002	floating vector
000001	fixed vector.

Field 5: handler prefix (4 characters maximum).

Field 6: device address size (decimal).

Field 7: major device number for block-type device.

Field 8: major device number for character-type device.

Field 9: maximum number of devices per controller (decimal).

Field 10: maximum bus request level (4 through 7).

Fields 11-13: optional configuration table structure declarations (8 characters maximum).

Part two contains lines with two fields each, as follows:

Field 1: alias of device (8 characters maximum).

Field 2: reference name of device (8 characters maximum, must have occurred in part one).

SEE ALSO

config(VIII)

NAME

mnttab – mounted file system table

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices mounted by the *mount*(VIII) command.

Each entry is 26 bytes in length; the first 10 bytes are the null-padded name of the place where the *special file* is mounted; the next 10 bytes represent the null-padded root name of the mounted special file; the remaining 6 bytes contain the mounted *special file's* read/write permissions and the date which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter, **NMOUNT**, located in **/sys/sys/cf/conf.c** which defines the number of allowable mounted special files.

SEE ALSO

mount(VIII), umount(VIII)

NAME

passwd – password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
comment
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The comment field should identify the user, e.g., <dept #> name (account #). Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

FILES

/etc/passwd

SEE ALSO

login(I), crypt(III), passwd(I), group(V)

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(III), and are interpreted for various devices by commands described in *plot*(I). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(III).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. Not effective in *vt0*. See *plot*(I).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise. Effective only in *vt0*.

c circle: The first four bytes give the center of the circle, the next two the radius. Effective only in *vt0*.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *tek*.

d dot: Begin a horizontal dotted line at the point given by the next four bytes. The following two bytes are a signed x-increment, and the two after are a word count. The indicated number of words follow. A point is plotted for each 1-bit in the list, and skipped for each 0-bit. Each point is offset rightward by the x-increment. Effective only in *vt0*.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(I). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

```
tek     space(0, 3120, 0, 3120);
t300    space(0, 4096, 0, 4096);
t300s   space(0, 4096, 0, 4096);
t450    space(0, 4096, 0,4096)
vt0     space(0, 2048, 0, 2048);
```

SEE ALSO

plot(I), plot(III), graph(I)

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as "the control character" and will be represented graphically as "@". Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form "DDDDD" represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum. The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The "@h" provides a "magic number" of (octal) 064001.

Delta table. The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
  .
  .
  .
@c <comments> ...
  .
  .
  .
@e
```

The first line (@s) contains the number of lines inserted/deleted/unchanged respectively. The second line (@d) contains the type of the delta (currently, normal: 'D', and removed: 'R'), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the @c lines (optional) contain comments associated with the delta.

The @e line ends the delta table entry.

User names. The login names of users who may add deltas to the file, separated by newlines. The lines containing these login names are surrounded by the bracketing lines "@u" and "@U". An empty list of user names allows anyone to make a delta.

Flags. Keywords used internally. Each flag line takes the form:

@f <flag> <optional text>

There are, at present, only eight flags defined:

@f t <type of program>
@f v <program name>
@f i
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default-sid>

The "t" flag defines the replacement for the %Y% identification keyword. The "v" flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The "i" flag controls the warning/error aspect of the "No id keywords" message. When the "i" flag is not present, this message is only a warning; when the "i" flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the "b" flag is present the **-b** keyletter may be used on the *get* command to cause a branch in the delta tree. The "m" flag defines the first choice for the replacement text of the %M% identification keyword. The "f" flag defines the "floor" release; the release below which no deltas may be added. The "c" flag defines the "ceiling" release; the release above which no deltas may be added. The "d" flag defines the default SID to be used when none is specified on a *get* command.

Comments. Arbitrary text surrounded by the bracketing lines "@t" and "@T". The comments section typically will contain a description of the file's purpose.

Body. The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

get(I), delta(I), admin(I), prt(I)
SCCS/PWB User's Manual by L. E. Bonanni and A. L. Glasser.

NAME

sha – Shell accounting file

DESCRIPTION

The file */etc/sha* is used by each Shell to record command execution data. This information is *not* used for charging, but is helpful for system tuning, command design, and monitoring of user activity. For each command executed, the Shell writes a 32-byte record of the following form:

```
struct {
        char    command_name[8];
        char    login_name[6];
        char    tty_letter;
        char    user_id;
        long    date;
        long    real_time;
        long    cpu_time;
        long    system_time;
} sh_record;
```

The *command_name* gives the last (or only) component of a pathname. When an asynchronously-executed command terminates, the Shell can obtain times, but not the actual command name. In this case, '**gok' is used. The name '()' indicates the completion of a parenthesized subshell.

The type (and therefore volume) of data recorded in */etc/sha* can be controlled by setting file permission bits appropriately. If it cannot be opened for writing, no data is recorded. Otherwise, the Shell tests the 3 bits of the group permission field to determine the kinds of recording to be done. If a Shell is reading from a TTY, it tests the high-order bit (04). If it is 0, the Shell records only external commands, i.e., those not built into the Shell. If the bit is 1, internal commands (such as *chdir*, =, etc.) are also recorded. A Shell that is not reading from a TTY uses the two low-order bits. If bit 02 is on, external commands are recorded. Setting bit 01 on adds internal commands. *Adm* should own */etc/sha*, and the group owner should be one not used elsewhere, such as 0. No data is ever recorded for the super-user. Sample file modes and their effects are:

606 Record external commands issued at TTY. This is the preferred mode.

666 Record everything but procedure-level internal commands, which can account for 30% of all command executions.

676 Record everything. This mode is probably of interest only to those who maintain the Shell. Be warned that this mode may cause */etc/sha* to grow by 1000 blocks per day in an active system.

SEE ALSO

sh(I), lastcom(VIII), sa(VIII)

NAME

tp – mag tape format

DESCRIPTION

The command *tp* dumps files to and extracts files from magtape.

Block zero contains a copy of a stand-alone bootstrap program. See *tapeboot(VIII)*.

Blocks 1 through 62 contain a directory of the tape. There are 496 entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

path name	32 bytes
mode	2 bytes
uid	1 byte
gid	1 byte
unused	1 byte
size	3 bytes
time modified	4 bytes
tape address	2 bytes
unused	16 bytes
check sum	2 bytes

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (fs(V)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks 63 on are available for file storage.

A fake entry has a size of zero. See *tp(I)*.

SEE ALSO

fs(V), tapeboot(VIII), tp(I)

NAME

ttys – terminal initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which terminal special files are to have a process created for them which will allow people to log in. It consists of lines of 3 characters each.

The first character is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is the last character in the name of a terminal; e.g. *x* refers to the file '/dev/tty*x*'. The third character is used as an argument to the *getty* program, which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (Getty will have t be fixed in such cases.)

FILES

/etc/ttys

SEE ALSO

init(VIII), getty(VIII), login(I)

NAME

utmp – user information

DESCRIPTION

This file allows one to discover information about who is currently using UNIX. The file is binary; each entry is 16(10) bytes long. The first eight bytes contain a user's login name or are null if the table slot is unused. The low order byte of the next word contains the last character of a terminal name. The next two words contain the user's login time. The last word is unused.

FILES

/etc/utmp

SEE ALSO

init(VIII) and login(I), which maintain the file; who(I), which interprets it.

NAME

wtmp – user login history

DESCRIPTION

This file records all logins and logouts. Its format is exactly like *utmp*(V) except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name ~ indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names '|' and '}' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by *login(I)* and *init(VIII)*. Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by *ac(VIII)*.

FILES

/usr/adm/wtmp

SEE ALSO

utmp(V), login(I), init(VIII), ac(VIII), who(I)

NAME

azel – satellite predictions

SYNOPSIS

/usr/games/azel [**–d**] [**–l**] satellite1 [**–d**] [**–l**] satellite2 ...

DESCRIPTION

Azel predicts, in convenient form, the apparent trajectories of Earth satellites whose orbital elements are given in the argument files. If a given satellite name cannot be read, an attempt is made to find it in a directory of satellites maintained by the program's author. The **–d** option causes *azel* to ask for a date and read line 1 data (see below) from the standard input. The **–l** option causes *azel* to ask for the observer's latitude, west-longitude, and height above sea level.

For each satellite given the program types its full name, the date, and a sequence of lines each containing a time, an azimuth, an elevation, a distance, and a visual magnitude. Each such line indicates that: at the indicated time, the satellite may be seen from Murray Hill (or provided location) at the indicated azimuth and elevation, and that its distance and apparent magnitude are as given. Predictions are printed only when the sky is dark (sun more than 5 degrees below the horizon) and when the satellite is not eclipsed by the earth's shadow. Satellites which have not been seen and verified will not have had their visual magnitude level set correctly. All times input and output by *azel* are GMT (Universal Time). The satellites for which elements are maintained are:

sla,b,e,f,k Skylab A through Skylab K. Skylab A is the laboratory; B was the rocket but it has crashed. A and probably K have been verified.

cop Copernicus I. Never verified.

oao Orbiting Astronomical Observatory. Seen and verified.

pag Pageos I. Seen and verified; fairly dim (typically 2nd-3rd magnitude), but elements are extremely accurate.

exp19 Explorer 19; seen and verified, but quite dim (4th-5th magnitude) and fast-moving.

c103b, c156b, c184b, c206b, c220b, c461b, c500b
Various of the USSR Cosmos series; none seen.

7276a Unnamed (satellite # 72-76A); not seen.

The element files used by *azel* contain 5 lines. The first line gives a year, month, day, hour, and minute at which the program begins its consideration of the satellite, followed by a number of minutes and an interval in minutes. If the year, month, and day are 0, they are taken to be the current date (taken to change at 6 A.M. local time). The output report starts at the indicated epoch and prints the position of the satellite for the indicated number of minutes at times separated by the indicated interval. This line is ended by 2 numbers that specify options to the program governing the completeness of the report; they are ordinarily both '1'; the first suppresses output when the sky is not dark; the second suppresses output when the satellite is eclipsed by the earth's shadow. The next line of an element file is the full name of the satellite. The next 3 are the elements themselves (including certain derivatives of the elements).

FILES

/usr/jfo/* – orbital element files

SEE ALSO

sky(VI)

NAME

bio – biorhythm analysis

SYNOPSIS

/usr/games/bio birth-date start-date [number-of-days]

DESCRIPTION

Bio produces a graph of a person's biorhythm functions. The date arguments are given in the form mm/dd/yy. The *number-of-days* argument is the number of days for which the graph is printed. The default is 30 days. The three biorhythm curves are plotted: physical (p), emotional (e), and intellectual (i).

To get a neatly formatted graph quickly use:

bio birth-date start-date ^ reform ^ pr ^ gsi

NAME

bj – the game of black jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

Bj is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is $2 every hand.

A player 'natural' (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a 'push' (no money exchange).

If the dealer has an ace up, the player is allowed to make an 'insurance' bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.

If the player is dealt two cards of the same value, he is allowed to 'double'. He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)

If a dealt hand has a total of ten or eleven, the player may 'double down'. He may double the bet ($2 to $4) and receive exactly one more card on that hand.

Under normal play, the player may 'hit' (draw a card) as long as his total is not over twenty-one. If the player 'busts' (goes over twenty-one), the dealer wins the bet.

When the player 'stands' (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.

If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by y followed by a new line for 'yes', or just new line for 'no'.

? (means, "do you want a hit?")
Insurance?
Double down?

Every time the deck is shuffled, the dealer so states and the 'action' (total bet) and 'standing' (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

NAME

chess – the game of chess

SYNOPSIS

/usr/games/chess

DESCRIPTION

Chess is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '+' is used to specify check; 'o-o' and 'o-o-o' specify castling. To play black, type 'first'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

FILES

/usr/lib/book opening 'book'

DIAGNOSTICS

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

WARNING

Over-use of this program will cause it to go away.

BUGS

Pawns may be promoted only to queens.

NAME

cubic – three dimensional tic-tac-toe

SYNOPSIS

/usr/games/cubic

DESCRIPTION

Cubic plays the game of three dimensional 4×4×4 tic-tac-toe. Moves are given by the three digits (each 1-4) specifying the coordinate of the square to be played.

WARNING

Too much playing of the game will cause it to disappear.

NAME

factor – discover prime factors of a number

SYNOPSIS

/usr/games/factor

DESCRIPTION

When *factor* is invoked, it prompts for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2×10^{16}) it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime. It takes 1 minute to factor a prime near 10^{13}.

DIAGNOSTICS

"Ouch." for input out of range or for garbage input.

NAME

moo – guessing game

SYNOPSIS

/usr/games/moo

DESCRIPTION

Moo is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A 'cow' is a correct digit in an incorrect position. A 'bull' is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

NAME

othello – a game of dramatic reversals

SYNOPSIS

/usr/games/othello [[**–r**] *file*]

DESCRIPTION

Othello (a.k.a *reversi*) is played on an 8 by 8 board using two-sided tokens. Each player takes his turn by placing a token with his side up in an empty square. During the first four turns, players may only place tokens in the four central squares of the board. Subsequently, with each turn, a player *must* capture one or more of his opponents tokens. He does this by placing one of his tokens such that he outflanks one or more of his opponents', horizontally, vertically or diagonally. Captured tokens are flipped over and thus can be re-captured. If a player cannot outflank his opponent he must forfeit his turn. The play continues until the board is filled or until no more outflanking is possible.

In this game, your tokens are asterisks and the machine's are at-signs. You move by typing in the row and column at which you want to place your token as two digits (1-8), optionally separated by blanks or tabs. You can also type in

- **c** to continue the game after hitting break (this is only necessary if you interrupt the machine while it is deliberating),
- **g** *n* to start *othello* playing against itself for the next *n* moves (or until the break key is hit),
- **n** to stop printing the board after each move,
- **o** to start it up again,
- **p** to print the board regardless,
- **q** to quit (without dishonor),
- **s** to print the score, and, as always,
- **!** to escape to the shell. Control-D gets you back.

Othello also recognizes several commands which are valid only at the start of the game, before any moves have been made. They are

- **f** to let the machine go first.
- **h** *n* to ask for a handicap of from one to four corner squares. If you're *really* good, you can give the machine a handicap by typing a negative number.
- **l** *n* to set the amount of lookahead used by the machine in searching for moves. Zero means none at all. Four is the default. Greater than six means you may fall asleep waiting for the machine to move.
- **t** *n* to tell *othello* that you will only need *n* seconds to consider each move. If you fail to respond in the allotted time, you forfeit your turn.

If *othello* is given a file name as an argument, it will checkpoint the game, move by move, by dumping the board onto *file*. The **–r** flag will cause *othello* to restart the game from *file* and continue logging.

DIAGNOSTICS

Illegal! and Huh?

NAME

sky – obtain ephemerides

SYNOPSIS

/usr/games/sky [–l]

DESCRIPTION

Sky predicts the apparent locations of the Sun, the Moon, the planets out to Saturn, stars of magnitude at least 2.5, and certain other celestial objects. *Sky* reads the standard input to obtain a GMT time typed on one line with blanks separating year, month number, day, hour, and minute; if the year is missing the current year is used. If a blank line is typed the current time is used. The program prints the azimuth, elevation, and magnitude of objects which are above the horizon at the ephemeris location of Murray Hill at the indicated time. The –l flag causes it to ask for another location.

Placing a "1" input after the minute entry causes the program to print out the Greenwich Sidereal Time at the indicated moment and to print for each body its topographic right ascension and declination as well as its azimuth and elevation. Also, instead of the magnitude, the semidiameter of the body, in seconds of arc, is reported.

A "2" after the minute entry makes the coordinate system geocentric.

The effects of atmospheric extinction on magnitudes are not included; the brightest magnitudes of variable stars are marked with "*".

For all bodies, the program takes into account precession and nutation of the equinox, annual (but not diurnal) aberration, diurnal parallax, and the proper motion of stars. In no case is refraction included.

The program takes into account perturbations of the Earth due to the Moon, Venus, Mars, and Jupiter. The expected accuracies are: for the Sun and other stellar bodies a few tenths of seconds of arc; for the Moon (on which particular care is lavished) likewise a few tenths of seconds. For the Sun, Moon and stars the accuracy is sufficient to predict the circumstances of eclipses and occultations to within a few seconds of time. The planets may be off by several minutes of arc.

There are lots of special options not described here, which do things like substituting named star catalogs, smoothing nutation and aberration to aid generation of mean places of stars, and making conventional adjustments to the Moon to improve eclipse predictions.

For the most accurate use of the program it is necessary to know that it actually runs in Ephemeris time.

FILES

/usr/lib/startab, /usr/lib/moontab

SEE ALSO

azel (VI)

American Ephemeris and Nautical Almanac, for the appropriate years; also, the *Explanatory Supplement to the American Ephemeris and Nautical Almanac*.

NAME

ttt – the game of tic-tac-toe

SYNOPSIS

/usr/games/ttt

DESCRIPTION

Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

FILES

/usr/games/ttt.k learning file

NAME

wump — the game of hunt-the-wumpus

SYNOPSIS

/usr/games/wump

DESCRIPTION

Wump plays the game of "Hunt the Wumpus." A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company, 2,* 2 (November 1973).

BUGS

It will never replace Space War.

NAME

terminals – descriptions of commonly-used terminals

DESCRIPTION

This page serves as an introduction and index to the pages in Section VII that describe some of the terminals in common use. These pages should help solve those problems that may occur during the actual use of the terminals. Note that no conclusions regarding terminal selection should be drawn from the presence or absence of specific terminals in these pages. Headings on these pages include:

COMMANDS TO ISSUE AFTER LOGIN – this section gives the commands necessary to properly initialize the state of the terminal. The commands usually include *tabs(I)* to set hardware tab stops, and *stty(I)* to set appropriate carriage return and line feed delays.

NORMAL SWITCH SETTINGS – this section notes the required settings for the various switches and toggles of the terminal. It is especially important to be aware of these when using a terminal in a public terminal room, as switches may be left in an unexpected setting, leading to odd results.

SPECIAL CHARACTERS AND STATES – characters having atypical effects are noted here, along with escape sequences that may be needed to generate useful actions.

COMMON PROBLEMS – this section lists problems commonly found when using the terminal and indicates possible remedies for them.

IDIOSYNCRASIES – notes unusual properties of the terminal.

Although almost any full-duplex ASCII terminal can be used with PWB/UNIX, some are much more suitable than others. Because the whole terminal situation changes rapidly, no recommendations are given here regarding choice of terminals.

SEE ALSO

stty(I), tabs(I)
DASI450(VII), GSI300(VII), HP2640(VII), TERMINET(VII), TI700(VII)

NAME

DASI450 – DASI450, DIABLO 1620, XEROX 1700 terminals

DESCRIPTION

The DASI450 is a useful general-purpose terminal, often used in document production. The primary advantages of this terminal include its wide variety of features, availability of many type fonts, high print quality, and ease of changing the print element and ribbon.

The terminal normally produces output 10 or 12 characters to the inch horizontally, allowing total line widths of 132 and 158 characters, respectively. Horizontal spacing is normally controlled by the SPACING switch (see below), but the setting of that switch can be *dynamically overridden* by appropriate control sequences, either from the keyboard or remotely. Vertical spacing is normally 6 lines per inch, and is independent of the horizontal spacing. Vertical spacing can be changed dynamically to 8 lines per inch and back to 6 by (different) control sequences. Using *graphics mode*, the print mechanism may be spaced in horizontal increments of 1/60 inch, and vertical increments of 1/48 inch. Combined with forward and reverse motions, *graphics mode* can be used to produce subscripts, superscripts, reverse line motion, Greek letters, and graphs. Output filters may be necessary for some of these functions: see *450(1)* and *graph(1)*. Graphics mode in entered or left by control sequences that can be generated dynamically, from the terminal or remotely (see COMMON PROBLEMS below).

COMMANDS TO ISSUE AFTER LOGIN

tabs +t450; stty nl0 cr2

This makes sure that tab stops are set. It also sets terminal delays appropriate for most output, especially that containing many contiguous blank lines. At this setting, it takes about 49 seconds per page of C program, and 84 seconds per page of *nroff(1)* output (UNIX manual page). A few rare types of output may not print properly at this setting. Usable settings and their approximate relative time ratios are as follows:

nl0 cr2	1.00
nl0 cr1	1.03
nl0 cr3	1.08
nl2 cr2	1.10
nl2 cr3	1.17

For output with many blank lines, the **cr2** and **cr3** settings seem to work best.

NORMAL SWITCH SETTINGS

Switches are inside the terminal cover, just above the keyboard. From left to right, they should be set as follows:

FORM LENGTH – 11
SPEED – 30
SPACING – 10 (or 12: see below; see also DESCRIPTION above.)
AUTO LF – OFF
PARITY – EVEN
DUPLEX – FULL

Switches at the lower left side of the keyboard should be set as shown:

LOCAL – not depressed
UC ONLY – normally not depressed

The switches at the upper right side of the keyboard should be set:

ERROR RESET – push this when red light at left goes on
FORM FEED – push to jump to top of form, as set by next button
SET TOF – at start of session, align paper to perforation, then push
SCROLL – normally OFF, although you may want to experiment with ON
POWER – ON

In 10-pitch mode, output is printed 10 characters/inch horizontally, 6 lines/inch vertically, so that a character is 6 plot increments wide, and 8 (vertical) plot increments high. This mode permits about 65 characters per line, 66 lines/page on normal 8 1/2" by 11" paper. This output size is compatible with many other terminals, and is expected as a default by many UNIX commands, such as *nroff(1)* and *pr(1)*. For normal output, the following are appropriate:

nroff −h −T450 *file...* or **nroff −h** *file...* **| 450**

In 12-pitch mode, output is printed 12 characters/inch, 6 lines/inch, so that a character is 5 increments wide and 8 high. This mode allows about 80 characters/line. The 12-pitch, 6 lines/inch combination is considered by many to be the most attractive output format. Use:
nroff −h −T450 −12 *files...* or **nroff −h** *files...* **| 450**

SPECIAL CHARACTERS AND STATES

The interrupt signal can be generated by hitting either the DEL or BREAK key; the former is usually more convenient. At any point in time, a terminal is either in *graphics mode* or *character mode*, and the interpretations of some characters differ according to mode. In *graphics mode*, it is possible to space a single increment in each direction.

COMMON PROBLEMS

OUTPUT GENERATED IN ONE POSITION, OVERPRINTING – you may accidentally have gotten into *graphics mode*. Type ESC followed by '4' to leave that mode.

GARBAGE OUTPUT, WITH WILD SKIPPING – a DASI may go berserk when faced with many very long lines, long sequences of nonblank, nonidentical characters requiring extreme print wheel motion, or heavy amounts of tabbing. Remove some tab characters or increase terminal delays via *stty*.

PRINT HEAD ZOOMS TO RIGHT SIDE OF CARRIAGE – tab stops are not set. Set them with the *tabs* command.

POOR REGISTRATION AFTER REVERSE PLATEN MOTION – this is most likely to occur when using a forms tractor to perform reverse line feeds or half-line motions. Some (but not all) forms tractors have just enough slack in their mechanism that it is difficult to return exactly to the position you want. For best appearance of such text, or of Greek letters, take the forms tractor off, and use the friction feed instead. This problem is very dependent on the individual terminal.

NO LINE FEED OCCURS WHEN RETURN HIT; NO SYSTEM RESPONSE TO RETURN – you are in a mode where there is no conversion of RETURN to CR-LF echoed to your terminal.

There are two situations. First, either the terminal or coupler switch may be set to HALF-DUPLEX, and you may have asked to suppress echoing because you were getting double characters. Change the switches to FULL-DUPLEX, and issue a **stty echo** command. The second case is that a **stty nl** command has been done, or some equivalent action, such as using LINE FEED rather than RETURN during your login sequence. Issue the command **stty −nl**, but terminate it with a LINE FEED, not a RETURN. This will restore the terminal to the normal state, allowing convenient use of RETURN again.

ERROR LIGHT ON, OTHER PECULIAR BEHAVIOR − push the RESET button found at the upper right side of the keyboard. If this does not help, take the cover off and push the CLEAR button at the extreme right. This resets the microprocessor, leaves graphics mode, clears all tabs, and returns the carriage. Then issue *tabs* command to reset the tabs. The error light also turns on if either you or the computer attempt to print while the front cover is off.

IDIOSYNCRASIES

A DASI can perform a high-speed skip when it receives a series of LF characters without other characters intermixed. Unfortunately, a newline is normally a CR-LF pair, and the terminal does not know that it is at the left margin, so that it does sequences of these pairs about 3 times slower than it needs to. As a result, the only way to assure high-speed skipping is to write code to convert a sequence of newlines into a single CR, followed by a sequence of LF's. PWB/UNIX does this under *stty* modes **nl0 cr2** and **nl0 cr3.**

SEE ALSO

450(I), graph(I), stty(I), tabs(I), terminals(VII)

NAME

GSI300 – GSI300 (DTC300 or DASI300) hard-copy terminals

DESCRIPTION

The GSI300 is a useful general-purpose terminal, often used in document production, although it is being supplanted by the newer DASI450 (DIABLO 1620 or XEROX 1700). The advantages of this terminal include its wide variety of features, availability of many type fonts, high print quality, and ease of changing the print element and ribbon.

The terminal can produce output at 10 or 12 characters to the inch horizontally, allowing total line widths of 132 and 158 characters, respectively. Vertical spacing can be set to 6 or 8 lines per inch. Both of these settings are under the exclusive control of the PITCH switch (see below). Using *plot mode*, the print mechanism may be spaced in horizontal increments of 1/60 inch, and vertical increments of 1/48 inch. Combined with forward and reverse motions, *plot mode* can be used to produce subscripts, superscripts, reverse line motion, Greek letters, and graphs. Output filters may be necessary for these functions: see *gsi(1)* and *graph(1)*. To use the plot mode, the PLOT switch must be ON (see below); once that switch is on, plot mode in entered or left by control sequences that can be generated dynamically, from the terminal or remotely (see COMMON PROBLEMS below).

COMMANDS TO ISSUE AFTER LOGIN

tabs; stty nl0 cr2

This makes sure that tab stops are set. It also sets terminal delays appropriate for most output, especially that containing many contiguous blank lines. At this setting, it takes about 49 seconds per page of C program, and 84 seconds per page of *nroff(1)* output (UNIX manual page). Some types of output may not print properly at this setting. Usable settings and their approximate relative time ratios are as follows:

nl0 cr2	1.00
nl0 cr1	1.03
nl0 cr3	1.08
nl2 cr2	1.10
nl2 cr3	1.17

For output with many blank lines, the **cr2** and **cr3** settings seem to work best; **nl2 cr3** is the safest choice for printing many consecutive lines of blankless text.

NORMAL SWITCH SETTINGS

Switches are inside the terminal cover, just above the keyboard. From left to right, they should be set as follows:

PARITY – EVEN
CODE – ASCII (if switch can be moved; it is a dummy on many terminals)
PLOT – ON (if present: some older terminals don't have one)
DUPLEX – FULL (if acoustic coupler is used, it should also be set to FULL)
BAUD – 300 (i.e., 30 characters per second)
PITCH – 10 (or 12: see below)
AUTO L.F. – OFF

At the lower left side of the keyboard, the LINE half of the LINE/LOCAL switch must be lit.

The PITCH switch controls both vertical and horizontal spacing in a coupled fashion. In 10-pitch mode, output is printed 10 characters/inch horizontally, 6 lines/inch vertically, so that a character is 6 plot increments wide, and 8 (vertical) plot increments high. This mode permits about 65 characters per line, 66 lines/page on normal 8 1/2" by 11" paper. This output size is compatible with many other terminals, and is expected as a default by many UNIX commands, such as *nroff(1)* and *pr(1)*. For normal output, the following are appropriate:

nroff −h −T300 *file...* or **nroff −h** *files...* **| gsi**

In 12-pitch mode, output is printed 12 characters/inch, 8 lines/inch, so that a character is 5 increments wide and only 6 high. This mode allows about 80 characters/line, 88 lines/page on the same size paper. Text printed 8 lines/inch appears almost unreadable, but this mode is a useful paper-saver for dumping files for reference. For example, use:

pr −l88 *file...*

to produce condensed listings.

The 12-pitch, 6 lines/inch combination is considered by many to be the most attractive output format. It is obtained by setting the PITCH switch to 12, the PLOT switch ON, and using:

nroff −h −T300−12 *file...* or **nroff −h** *file...* **| gsi +12**

SPECIAL CHARACTERS AND STATES

The interrupt signal can be generated by hitting either the DEL or BREAK key; the latter is usually more convenient, being independent of the SHIFT key. At any point in time, a terminal is either in *plot mode* or *character mode*, and the interpretation of some characters differs according to mode. If the PLOT switch is ON, the BEL character (octal 006, CONTROL "g" on terminal) changes the mode to *character mode*, and the ACK character (octal 007, CONTROL "f" on terminal) changes the mode from the current mode to the other one. If the PLOT switch is OFF, the terminal is always in *character mode*. In *plot mode*, it is possible to space a single increment in each direction. Useful motion characters include the following:

SP (space, octal 040) − 1/60" right
BS (backspace, octal 010) − 1/60" left
LF (line feed, octal 012) − 1/48" forward
VT (reverse line feed for this terminal, octal 013) − 1/48" backwards

COMMON PROBLEMS

OUTPUT GENERATED IN ONE POSITION, OVERPRINTING − you may accidentally have gotten into **plot mode.** Hold CONTROL down while hitting "g", producing a BEL character to leave that mode.

GARBAGE OUTPUT, WITH WILD SKIPPING − a GSI may go berserk when faced with many very long lines, long sequences of non-blank, non-identical characters requiring extreme print wheel motion, or heavy amounts of tabbing. The GSI's microprocessor exceeds its 128-character buffer and becomes very confused. Remove some tab characters, use the *gsi* command's delay option, or increase terminal delays via *stty*.

PRINT HEAD ZOOMS TO RIGHT SIDE OF CARRIAGE – tab stops are not set. Set them with the *tabs* command.

POOR REGISTRATION AFTER REVERSE PLATEN MOTION – this is most likely to occur when using a forms tractor to perform reverse line feeds or half-line motions. Some (but not all) forms tractors have just enough slack in their mechanism that it is difficult to return exactly to the position you want. For best appearance of such text, or of Greek letters, take the forms tractor off, and use the friction feed instead. This problem is very dependent on the individual terminal.

NO LINE FEED OCCURS WHEN RETURN HIT; NO SYSTEM RESPONSE TO RETURN – you are in a mode where there is no conversion of RETURN to CR-LF echoed to your terminal. There are two situations. First, either the terminal or coupler switch may be set to HALF-DUPLEX, and you may have asked to suppress echoing because you were getting double characters. Change the switches to FULL-DUPLEX, and issue a **stty echo** command. The second case is that a **stty nl** command has been done, or some equivalent action, such as using LINE FEED rather than RETURN during your login sequence. Issue the command **stty −nl**, but terminate it with a LINE FEED, not a RETURN. This will restore the terminal to the normal state, allowing convenient use of RETURN again.

FAULT LIGHT ON, OTHER PECULIAR BEHAVIOR – push the RESET button found under the right side of the cover. This resets the microprocessor, gets out of plot mode, clears all tabs, and returns the carriage. Then issue *tabs* command to reset the tabs.

IDIOSYNCRASIES

A GSI can perform a high-speed skip when it receives a series of LF characters without other characters intermixed. Unfortunately, a newline is normally a CR-LF pair, and the terminal does not know that it is at the left margin, so that it does sequences of these pairs about 3 times slower than it needs to. As a result, the only way to assure high-speed skipping is to write code to convert a sequence of newlines into a single CR, followed by a sequence of LF's. PWB/UNIX does this under *stty* modes **nl0 cr2** and **nl0 cr3**.

SEE ALSO

gsi(I), graph(I), stty(I), tabs(I), terminals(VII)

NAME

HP2640 – Hewlett-Packard 2640 CRT terminal family

DESCRIPTION

This family contains a large and growing number of models that appear to be similar, but have slight variations in keyboard layout and major variations in options and peripheral devices. The HP2640B appears to be the most popular model at the current time. It is suitable for both programming and documentation work. Positive features of the terminal include hardware tab stops, large local memory (up to 8K bytes) with convenient scanning, ability to lock several lines on the display, display enhancements which permit readable display of most *nroff(1)* output, and displayable graphics for control characters.

Quick perusal of *nroff* output can be obtained using the *hp(1)* filter:

nroff –h *options files...* **| hp**

COMMANDS TO ISSUE AFTER LOGIN

tabs +thp; stty nl0 cr0

This sequence sets UNIX standard tab stops (every 8 columns), then turns off (unnecessary) line feed and carriage return delays.

NORMAL SWITCH SETTINGS

The ON/OFF switch is at the left rear of the terminal. The following switches are at the upper left of the keyboard:

DUPLEX – FULL
PARITY – EVEN
BAUD RATE – 300 (1200 could be used with proper modem)
BLOCK MODE – not depressed
REMOTE – depressed
CAPS LOCK – not depressed (for most uses)
MEMORY LOCK – not depressed unless lines are to be locked on screen
AUTO LF – not depressed

None of the other latching keys should be depressed.

SPECIAL CHARACTERS AND STATES

An interrupt can be generated by DEL or BREAK. The location of the BREAK key varies among models.

COMMON PROBLEMS

NO LINE FEED OCCURS WHEN RETURN HIT; NO SYSTEM RESPONSE TO RETURN – you are in a mode where there is no conversion of RETURN to CR-LF echoed to your terminal. There are two situations. First, either the terminal or coupler switch may be set to HALF-DUPLEX, and you may have asked to suppress echoing because you were getting double characters. Change the switches to FULL-DUPLEX, and issue a **stty echo** command. The second case is that a **stty nl** command has been done, or some equivalent action, such as using LINE FEED rather than RETURN during your login sequence. Issue the command **stty –nl**, but terminate it with a LINE FEED, not a RETURN. This will restore the terminal to the normal state, allowing

convenient use of RETURN again.

If the terminal does not seem to work, try the RESET button. Note that this action clears tab stops.

IDIOSYNCRASIES

When the terminal receives a Horizontal Tab character that occurs beyond the last tab stop (if any), the effect is that of a newline. Thus, *tabs(I)* may cause rapid scrolling while clearing tabs.

SEE ALSO

hp(I), stty(I), tabs(I), terminals(VII)

NAME

TermiNet – GE TermiNet 300 (and 1200) terminals

DESCRIPTION

The TermiNet 300 is a reasonable terminal for general-purpose use. Because it does provide hardware tab stops, it is useful for both programming and documentation. It prints up to 118 characters (10-pitch), utilizing a continuously-moving band of print elements. The terminal is reasonably compact. A useful feature is the fact that the first tab stop set on the terminal becomes the left margin. Some users prefer this terminal's column position lights and lack of large moving print element. Visibility of current typed line is adequate.

The TermiNet 1200 is a 1200-baud version of the 300.

COMMANDS TO ISSUE AFTER LOGIN

tabs or **tabs +ttn**

This assures setting of UNIX standard tab stops. By default, you also have delays set as by stty **nl0 cr1**, which should generally work, but may fail for some types of output. On a TermiNet 300 at this setting, it takes about 49 seconds per page of C program, and 84 seconds per page of *nroff* output (UNIX manual page), the latter figure assuming output is tabbed. Usable settings and their relative time ratios are as follows:

nl0 cr2	1.00
nl0 cr1	1.03
nl0 cr3	1.08
nl2 cr2	1.10
nl2 cr3	1.17

The TermiNet 1200 is about 2.5 times faster than the 300 at corresponding settings, but may not be able to print properly at the fastest settings.

NORMAL SWITCH SETTINGS

Several switches are on back of the terminal:

(Back left) – NORM (not CAPS ONLY)
(Back left, on some models) – FULL DUPLEX
(Back right) – power ON

Light switches on front left:

ON LINE – push so that it becomes lit
INTERRUPT – if lit, push it so it goes out

Switches on right front:

TRANSPARENCY – OFF
INHIBIT – NORM
RATE – 30
LINE FEED – 1
AUTO L.F. – OFF

COMMON PROBLEMS

NO LINE FEED OCCURS WHEN RETURN HIT; NO SYSTEM RESPONSE TO RETURN – you are in a mode where there is no conversion of RETURN to CR-LF echoed to your terminal. There are two situations. First, either the terminal or coupler switch may be set to HALF-DUPLEX, and you may have asked to suppress echoing because you were getting double characters. Change the switches to FULL-DUPLEX, and issue a **stty echo** command. The second case is that a **stty nl** command has been done, or some equivalent action, such as using LINE FEED rather than RETURN during your login sequence. Issue the command **stty −nl**, but terminate it with a LINE FEED, not a RETURN. This will restore the terminal to the normal state, allowing convenient use of RETURN again.

SEE ALSO

stty(I), tabs(I), terminals(VII)

NAME

TI700 – TI 745, 735, and 725 terminals

DESCRIPTION

The TI745 (and to a lesser extent, the TI735) are lighter and smaller than the older TI725, and their keyboards are more suitable for general-purpose UNIX usage. In particular, the DEL key and backslash are favorably placed, and they provide an underscore in place of the 725's back-arrow. *Nroff(1)* output is thus more readable on the 745 and 735. Output is printed on thermal paper, with a carriage width of 80 characters. The TI745 accepts a smaller roll of paper than the others, but is much more portable.

COMMANDS TO ISSUE AFTER LOGIN

stty –tabs nl0 cr2

This requests UNIX to simulate standard UNIX tab stops (every 8 columns). It also lessens carriage return and line feed delays to the minimum acceptable to the terminal. If the terminal cannot print something at this setting, various other settings may be tried. At the **nl0 cr2** setting, it takes about 65 seconds per page of C program, and 93 seconds per page of *nroff* output (UNIX Manual page). Usable settings and their relative time ratios are as follows:

nl0 cr2	1.00
nl0 cr1	1.03
nl0 cr3	1.06
nl2 cr2	1.08
nl2 cr3	1.14

The lack of hardware tabs causes these terminals to require about 15-20% more time than 300-baud terminals providing tabs.

NORMAL SWITCH SETTINGS (745)

Most switches are right-left toggles in front of the keyboard.

UPPER CASE (left front) – right side depressed
HALF DUP (right front) – right side depressed
LOW SPEED (right front) – right side depressed
ON LINE (right front) – left side depressed
MARK-EVEN-ODD (right rear) – EVEN
ON-OFF (right rear) – toggle back

NORMAL SWITCH SETTINGS (735-725)

Most of the switches are on the upper left side of the terminal:

LINE FEED – SINGLE
SPEED – 30
DUPLEX – FULL
PARITY – EVEN
INTERFACE – INT

In addition, the PWR switch must of course be turned on, and the ON LINE switch depressed. You will be in local mode otherwise, and get no response whatsoever.

SPECIAL CHARACTERS

To generate a Horizontal Tab character from the keyboard, hold CTRL down and hit "i".

You can interrupt an executing program with either the DEL or BREAK keys.

COMMON PROBLEMS

NO LINE FEED OCCURS WHEN RETURN HIT; NO SYSTEM RESPONSE TO RETURN – you are in a mode where there is no conversion of RETURN to CR-LF echoed to your terminal. There are two situations. First, either the terminal or coupler switch may be set to HALF-DUPLEX, and you may have asked to suppress echoing because you were getting double characters. Change the switches to FULL-DUPLEX, and issue a **stty echo** command. The second case is that a **stty nl** command has been done, or some equivalent action, such as using LINE FEED rather than RETURN during your login sequence. Issue the command **stty −nl**, but terminate it with a LINE FEED, not a RETURN. This will restore the terminal to the normal state, allowing convenient use of RETURN again.

SEE ALSO

stty(I), terminals(VII)

NAME

tmac.name – standard nroff and troff macro packages

DESCRIPTION

A number of standard macro packages have been written for use with the UNIX text formatters, *nroff(I)* and *troff(I)*. When using either of these commands, an argument of the form **–m***name* requests inclusion of the file named **/usr/lib/tmac.***name*.

The following macro packages are supported by PWB/UNIX. All but the last can be used with either *nroff* or *troff*. The last one works with *troff* only.

Name	*Description/Documentation*
a	Same as /usr/man/man0/caa. See *PWB/UNIX Manual Page Macros* by E. M. Piskorik.
m	PWB/MM; a unified, general-purpose set of macros for memoranda, manuals, letters, etc. See *PWB/MM – Programmer's Workbench Memorandum Macros* by D. W. Smith and J. R. Mashey.
org	BTL organization chart macros.
uom	UNIX Operations Manual macros; uses –mm.
v	View graph and slide macros. See *PWB/UNIX View Graph and Slide Macros* by T. A. Dolotta and D. W. Smith.

SEE ALSO

nroff(I), troff(I)