-

# UNIX PROGRAMMER'S MANUAL

*Sixth Edition*

*K. Thompson*

*D. M. Ritchie*

*May, 1975*

-

This manual was set by a Graphic Systems phototypesetter driven by the *troff* formatting program operating under the UNIX system. The text of the manual was prepared using the *ed* text editor.

-

# PREFACE
to the Sixth Edition

We are grateful to L. L. Cherry, R. C. Haight, S. C. Johnson, B. W. Kernighan, M. E. Lesk, and E. N. Pinson for their contributions to the system software, and to L. E. McMahon for software and for his contributions to this manual. We are particularly appreciative of the invaluable technical, editorial, and administrative efforts of J. F. Ossanna, M. D. McIlroy, and R. Morris. They all contributed greatly to the stock of UNIX software and to this manual. Their inventiveness, thoughtful criticism, and ungrudging support increased immeasurably not only whatever success the UNIX system enjoys, but also our own enjoyment in its creation.

-

## INTRODUCTION TO THIS MANUAL

This manual gives descriptions of the publicly available features of UNIX. It provides neither a general overview – see ‘‘The UNIX Time-sharing System’’ (Comm. ACM **17** 7, July 1974, pp. 365-375) for that – nor details of the implementation of the system, which remain to be disclosed.

Within the area it surveys, this manual attempts to be as complete and timely as possible. A conscious decision was made to describe each program in exactly the state it was in at the time its manual section was prepared. In particular, the desire to describe something as it should be, not as it is, was resisted. Inevitably, this means that many sections will soon be out of date.

This manual is divided into eight sections:

| | |
|---|---|
| I. | Commands |
| II. | System calls |
| III. | Subroutines |
| IV. | Special files |
| V. | File formats and conventions |
| VI. | User-maintained programs |
| VII. | User-maintained subroutines |
| VIII. | Maintenance |

Commands are programs intended to be invoked directly by the user, in contradistinction to subroutines, which are intended to be called by the user’s programs. Commands generally reside in 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.

System calls are entries into the UNIX supervisor. In assembly language, they are coded with the use of the opcode *sys,* a synonym for the *trap* instruction. In this edition, the C language interface routines to the system calls have been incorporated in section II.

A small assortment of subroutines is available; they are described in section III. The binary form of most of them is kept in the system library */lib/liba.a.* The subroutines available from C and from Fortran are also included; they reside in */lib/libc.a* and */lib/libf.a* respectively.

The special files section IV discusses the characteristics of each system ‘‘file’’ which actually refers to an I/O device. The names in this section refer to the DEC device names for the hardware, instead of the names of the special files themselves.

The file formats and conventions section V documents the structure of particular kinds of files; for example, the form of the output of the loader and assembler is given. Excluded are files used by only one command, for example the assembler’s intermediate files.

User-maintained programs and subroutines (sections VI and VII) are not considered part of the UNIX system, and the principal reason for listing them is to indicate their existence without necessarily giving a complete description. The authors of the individual programs should be consulted for more information.

Section VIII discusses commands which are not intended for use by the ordinary user, in some cases because they disclose information in which he 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, its preparation date in the upper middle. Entries within each section are alphabetized. The page numbers of each entry start at 1. (The earlier hope for frequent, partial updates of the manual is clearly in vain, but in any event it is not feasible to maintain consecutive page numbering in a document like this.)

-

All entries are based on a common format, not all of whose subsections will always appear.

The *name* section repeats the entry name and gives a very short description of its purpose.

The *synopsis* summarizes the use of the program being described. A few conventions are used, particularly in the Commands section:

**Boldface** words are considered literals, and are typed just as they appear.

Square brackets ( [ ] ) around an argument indicate that the argument is optional. When an argument is given as ‘‘name’’, it always refers to a file name.

Ellipses ‘‘. . .’’ are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus sign ‘‘_’’ is often taken to mean 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 ‘‘_’’.

The *description* section discusses in detail the subject at hand.

The *files* section gives the names of files which are built into the program.

A *see also* section gives pointers to related information.

A *diagnostics* section discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.

The *bugs* section gives known bugs and sometimes deficiencies. Occasionally also the suggested fix is described.

At the beginning of this document is a table of contents, organized by section and alphabetically within each section. There is also a permuted index derived from the table of contents. Within each index entry, the title of the writeup to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands which exist only to exercise a particular system call.

This manual was prepared using the UNIX text editor *ed* and the formatting program *troff*.

-

## HOW TO GET STARTED

This section provides the basic information you need to get started on UNIX: how to log in and log out, how to communicate through your terminal, and how to run a program. See ''UNIX for Beginners'' by Brian W. Kernighan for a more complete introduction to the system.

*Logging in.* You must call UNIX from an appropriate terminal. UNIX supports ASCII terminals typified by the TTY 37, the GE Terminet 300, the Dasi 300, and various graphical terminals. You must also have a valid user name, which may be obtained, together with the telephone number, from the system administrators. The same telephone number serves terminals operating at all the standard speeds. After a data connection is established, the login procedure depends on what kind of terminal you are using.

> *300-baud terminals:* Such terminals include the GE Terminet 300, most display terminals, Execuport, TI, GSI, and certain Anderson-Jacobson terminals. These terminals generally have a speed switch which should be set at ''300'' (or ''30'' for 30 characters per second) and a half/full duplex switch which should be set at full-duplex. (This switch will often have to be changed since many other systems require half-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, the system asks for it and turns off the printer on the terminal so the password will not appear. After you have logged in, the ''return'', ''new line'', or ''linefeed'' keys will give exactly the same results.
>
> *TTY 37 terminal:* 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 the correct speed; you respond with your user name. From the TTY 37 terminal, and any other which has the ''new-line'' function (combined carriage return and line-feed), terminate each line you type with the ''new-line'' key (*not* the ''return'' key).

For all these terminals, 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 letters to lower case.

The evidence that you have successfully logged in is that the Shell program will type a ''%'' to you. (The Shell is described below under ''How to run a program.'')

For more information, consult *getty* (VIII), which discusses the login sequence in more detail, and *tty* (IV), which discusses typewriter I/O.

*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 (EOT character, 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 (I).

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

UNIX typewriter I/O 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 up 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 typewriter input line, the character ''@'' kills all the characters typed before it, so typing mistakes can be repaired on a single line. Also, 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 ‘‘\’’. (So, 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.* 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, 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 carriage-return and line-feed. In the latter case, all input carriage returns are turned to new-line characters (the standard line delimiter) and both a carriage return and a line feed are echoed to the terminal. If you get into the wrong mode, the *stty* command (I) 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 them turned into spaces during output, and echoed as spaces during input. The system assumes that tabs are set every eight columns. Again, the *stty* command (I) will set or reset this mode. Also, there is a file which, if printed on TTY 37 or TermiNet 300 terminals, will set the tab stops correctly (*tabs* (V)).

Section *tty* (IV) discusses typewriter I/O more fully.

*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. The Shell looks first in your current directory (see next section) for a program with the given name, and if none is there, then in a system directory. There is nothing special about system-provided commands except that they are kept in a directory where the Shell can find them.

The command name is always the first word on an input line; it and its arguments are separated from one another by spaces.

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 section *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 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 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 owners. As a matter of observed fact, few UNIX users 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 ‘‘/’’, 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 sub-

-

directory (no prefixed ‘‘/’’).

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

Important commands which 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; *rmdir* (I) for destroying them.

For a fuller discussion of the file system, see ‘‘The UNIX Time-Sharing System,’’ by the present authors. 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 in UNIX are assembly language (see *as* (I)), Fortran (see *fc* (I)), and C (see *cc* (I)). After the program text has been entered through the editor and written on a file, you can give the file to the appropriate language processor as an argument. The output of the language processor will be left on a file in the current directory named ‘‘a.out’’. (If the output is precious, use *mv* 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. The commands most often used to write text on a terminal are: *cat, pr, roff, nroff,* and *troff,* all in section I.

The *cat* command simply dumps ASCII text on the terminal, with no processing at all. The *pr* command paginates the text, supplies headings, and has a facility for multi-column output. *Troff* and *nroff* are elaborate text formatting programs, and require careful forethought in entering both the text and the formatting commands into the input file. *Troff* drives a Graphic Systems phototypesetter; it was used to produce this manual. *Nroff* produces output on a typewriter terminal. *Roff* (I) is a somewhat less elaborate text formatting program, and requires somewhat less forethought.

*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 next logs in. The write-ups in the manual also suggest how to respond to the two commands if you are a target.

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

TABLE OF CONTENTS

-

-

## PERMUTED INDEX

| | |
|---|---|
| dp(IV) DP-11 | 201 data-phone interface |
| | abort(III) generate an IOT fault |
| | abs, fabs(III) absolute value |
| abs, fabs(III) | absolute value |
| ac(VIII) login | accounting |
| sa(VIII) Shell | accounting |
| dn(IV) DN-11 | ACU interface |
| | ac(VIII) login accounting |
| shift(I) | adjust Shell arguments |
| primes(VI) print | all primes larger than somewhat |
| wall(VIII) write to | all users |
| | alloc, free(III) core allocator |
| salloc(VII) string | allocation and manipulation |
| break, brk, sbrk(II) change core | allocation |
| alloc, free(III) core | allocator |
| plot: openpl et | al.(VII) graphics interface |
| yacc(I) yet | another compiler-compiler |
| write(I) write to | another user |
| | a.out(V) assembler and link editor output |
| bc(I) | arbitrary precision interactive language |
| atan, atan2(III) | arc tangent function |
| ar(I) | archive and library maintainer |
| ar(V) | archive (library) file format |
| nargs(III) | argument count |
| getarg, iargc(III) get command | arguments from Fortran |
| echo(I) echo | arguments |
| glob(VIII) generate command | arguments |
| shift(I) adjust Shell | arguments |
| | ar(I) archive and library maintainer |
| | ar(V) archive (library) file format |
| ascii(V) map of | ASCII character set |
| atof(III) convert | ASCII to floating |
| atoi(III) convert | ASCII to integer |
| gmtime(III) convert date and time to | ASCII...ctime, localtime, |
| | ascii(V) map of ASCII character set |
| | as(I) assembler |
| a.out(V) | assembler and link editor output |
| as(I) | assembler |
| kl(IV) KL-11 or DL-11 | asynchronous interface |
| nice(I) run a command | at low priority |
| | atan, atan2(III) arc tangent function |
| atan, | atan2(III) arc tangent function |
| | atof(III) convert ASCII to floating |
| | atoi(III) convert ASCII to integer |
| wait(I) | await completion of process |
| | azel(VI) satellite predictions |
| | bas(I) basic |
| bas(I) | basic |
| | bc(I) arbitrary precision interactive language |
| su(VIII) | become privileged user |
| strip(I) remove symbols and relocation | bits |

-

| | |
|---|---|
| | bj(VI) the game of black jack |
| bj(VI) the game of | black jack |
| sync(VIII) update the super | block |
| update(VIII) periodically update the super | block |
| | boot procedures(VIII) UNIX startup |
| | break, brk, sbrk(II) change core allocation |
| break, | brk, sbrk(II) change core allocation |
| getc, getw, fopen(III) | buffered input |
| putc, putw, fcreat, fflush(III) | buffered output |
| mknod(VIII) | build special file |
| cc(I) | C compiler |
| cdb(I) | C debugger |
| dc(I) desk | calculator |
| cal(VI) print | calendar |
| indir(II) indirect system | call |
| intro(II) introduction to system | calls |
| | cal(VI) print calendar |
| ierror(III) | catch Fortran errors |
| signal(II) | catch or ignore signals |
| | cat(I) concatenate and print |
| | cat(IV) phototypesetter interface |
| | cc(I) C compiler |
| | cdb(I) C debugger |
| floor, | ceil(III) floor and ceiling functions |
| floor, ceil(III) floor and | ceiling functions |
| break, brk, sbrk(II) | change core allocation |
| chgrp(VIII) | change group |
| passwd(I) | change login password |
| chmod(II) | change mode of file |
| chmod(I) | change mode |
| chown(II) | change owner and group of a file |
| chown(VIII) | change owner |
| chdir(I) | change working directory |
| chdir(II) | change working directory |
| pipe(II) create an interprocess | channel |
| gsi(VI) interpret extended | character set on GSI terminal |
| ascii(V) map of ASCII | character set |
| getchar(III) read | character |
| putchar, flush(III) write | character |
| | chdir(I) change working directory |
| | chdir(II) change working directory |
| file system directory consistency | check...dcheck(VIII) |
| file system storage consistency | check...icheck(VIII) |
| chess(VI) the game of | chess |
| | chess(VI) the game of chess |
| | chgrp(VIII) change group |
| | chmod(I) change mode |
| | chmod(II) change mode of file |
| | chown(II) change owner and group of a file |
| | chown(VIII) change owner |
| clri(VIII) | clear i-node |
| cron(VIII) | clock daemon |
| close(II) | close a file |

-

| | |
|---|---|
| | close(II) close a file |
| | clri(VIII) clear i-node |
| | cmp(I) compare two files |
| | col(VI) filter reverse line feeds |
| getarg, iargc(III) get | command arguments from Fortran |
| glob(VIII) generate | command arguments |
| nice(I) run a | command at low priority |
| exit(I) terminate | command file |
| nohup(I) run a | command immune to hangups |
| sh(I) shell | (command interpreter) |
| goto(I) | command transfer |
| if(I) conditional | command |
| time(I) time a | command |
| | comm(I) print lines common to two files |
| comm(I) print lines | common to two files |
| dc(IV) DC-11 | communications interface |
| dh(IV) DH-11 | communications multiplexer |
| diff(I) differential file | comparator |
| cmp(I) | compare two files |
| cc(I) C | compiler |
| tmg(VI) | compiler-compiler |
| yacc(I) yet another | compiler-compiler |
| fc(I) Fortran | compiler |
| rc(I) Ratfor | compiler |
| wait(I) await | completion of process |
| cat(I) | concatenate and print |
| if(I) | conditional command |
| dcheck(VIII) file system directory | consistency check |
| icheck(VIII) file system storage | consistency check |
| csw(II) read | console switches |
| mkfs(VIII) | construct a file system |
| ls(I) list | contents of directory |
| init(VIII) process | control initialization |
| units(VI) | conversion program |
| ecvt, fcvt(III) output | conversion |
| locv(III) long output | conversion |
| dd(I) | convert and copy a file |
| atof(III) | convert ASCII to floating |
| atoi(III) | convert ASCII to integer |
| ctime, localtime, gmtime(III) | convert date and time to ASCII |
| dd(I) convert and | copy a file |
| cp(I) | copy |
| break, brk, sbrk(II) change | core allocation |
| alloc, free(III) | core allocator |
| core(V) format of | core image file |
| mem, kmem, null(IV) | core memory |
| | core(V) format of core image file |
| crread, crwrite, crexch, crprior(VII) | coroutine scheme...crfork, crexit, |
| sin, | cos(III) trigonometric functions |
| nargs(III) argument | count |
| wc(I) word | count |
| | cp(I) copy |
| crash(VIII) what to do when the system | crashes |

| | |
|---|---|
| | crash(VIII) what to do when the system crashes |
| creat(II) | create a new file |
| pipe(II) | create an interprocess channel |
| | creat(II) create a new file |
| | cref(I) make cross reference listing |
| crfork, crexit, crread, crwrite, | crexch, crprior(VII) coroutine scheme |
| scheme...crfork, | crexit, crread, crwrite, crexch, crprior(VII) coroutine |
| coroutine scheme... | crfork, crexit, crread, crwrite, crexch, crprior(VII) |
| | cron(VIII) clock daemon |
| cref(I) make | cross reference listing |
| crfork, crexit, crread, crwrite, crexch, | crprior(VII) coroutine scheme |
| crfork, crexit, | crread, crwrite, crexch, crprior(VII) coroutine scheme |
| crfork, crexit, crread, | crwrite, crexch, crprior(VII) coroutine scheme |
| | crypt(III) password encoding |
| | csw(II) read console switches |
| ASCII... | ctime, localtime, gmtime(III) convert date and time to |
| | cubic(VI) three dimensional tic-tac-toe |
| ttyn(III) return name of | current typewriter |
| spline(VI) interpolate smooth | curve |
| cron(VIII) clock | daemon |
| dpd(VIII) data phone | daemon |
| lpd(VIII) line printer | daemon |
| dpd(VIII) | data phone daemon |
| dp(IV) DP-11 201 | data-phone interface |
| prof(I) display profile | data |
| ttys(V) typewriter initialization | data |
| ctime, localtime, gmtime(III) convert | date and time to ASCII |
| time(II) get | date and time |
| date(I) print and set the | date |
| | date(I) print and set the date |
| | db(I) debug |
| dc(IV) | DC-11 communications interface |
| | dcheck(VIII) file system directory consistency check |
| | dc(I) desk calculator |
| | dc(IV) DC-11 communications interface |
| | dd(I) convert and copy a file |
| db(I) | debug |
| cdb(I) C | debugger |
| tp(V) | DEC/mag tape formats |
| tp(I) manipulate | DECtape and magtape |
| tc(IV) TC-11/TU56 | DECtape |
| dsw(I) | delete interactively |
| mesg(I) permit or | deny messages |
| dup(II) duplicate an open file | descriptor |
| mail(I) send mail to | designated users |
| dc(I) | desk calculator |
| file(I) | determine file type |
| | df(VIII) disk free |
| dh(IV) | DH-11 communications multiplexer |
| | dh(IV) DH-11 communications multiplexer |
| diff(I) | differential file comparator |
| | diff(I) differential file comparator |
| cubic(VI) three | dimensional tic-tac-toe |

| | |
|---|---|
| dir(V) format of | directories |
| dcheck(VIII) file system | directory consistency check |
| unlink(II) remove | directory entry |
| pwd(I) working | directory name |
| mknod(II) make a | directory or a special file |
| chdir(I) change working | directory |
| chdir(II) change working | directory |
| ls(I) list contents of | directory |
| mkdir(I) make a | directory |
| rmdir(I) remove | directory |
| | dir(V) format of directories |
| factor(VI) | discover prime factors of a number |
| hs(IV) RH11/RS03-RS04 fixed-head | disk file |
| rf(IV) RF11/RS11 fixed-head | disk file |
| df(VIII) | disk free |
| du(I) summarize | disk usage |
| hp(IV) RH-11/RP04 moving-head | disk |
| rk(IV) RK-11/RK03 (or RK05) | disk |
| rp(IV) RP-11/RP03 moving-head | disk |
| umount(II) | dismount file system |
| umount(VIII) | dismount file system |
| prof(I) | display profile data |
| ldiv, lrem(III) long | division |
| kl(IV) KL-11 or | DL-11 asynchronous interface |
| dn(IV) | DN-11 ACU interface |
| | dn(IV) DN-11 ACU interface |
| crash(VIII) what to | do when the system crashes |
| dp(IV) | DP-11 201 data-phone interface |
| | dpd(VIII) data phone daemon |
| | dp(IV) DP-11 201 data-phone interface |
| graph(VI) | draw a graph |
| | dsw(I) delete interactively |
| | du(I) summarize disk usage |
| dump(V) incremental | dump tape format |
| dump(VIII) incremental file system | dump |
| od(I) octal | dump |
| | dump(V) incremental dump tape format |
| | dump(VIII) incremental file system dump |
| | dup(II) duplicate an open file descriptor |
| dup(II) | duplicate an open file descriptor |
| echo(I) | echo arguments |
| | echo(I) echo arguments |
| | ecvt, fcvt(III) output conversion |
| end, etext, | edata(III) last locations in program |
| | ed(I) text editor |
| fed(VI) | edit form letter memory |
| a.out(V) assembler and link | editor output |
| ed(I) text | editor |
| ld(I) link | editor |
| crypt(III) password | encoding |
| | end, etext, edata(III) last locations in program |
| nlist(III) get | entries from name list |
| unlink(II) remove directory | entry |

-

| | |
|---:|:---|
| sky(VI) obtain | ephemerides |
| | eqn(I) typeset mathematics |
| perror, syserrlist, sysnerr, | errno(III) system error messages |
| sysnerr, errno(III) system | error messages...perror, syserrlist, |
| ierror(III) catch Fortran | errors |
| spell(I) find spelling | errors |
| plot: openpl | et al.(VII) graphics interface |
| end, | etext, edata(III) last locations in program |
| pfe(I) print floating | exception |
| | exec, execl, execv(II) execute a file |
| exec, | execl, execv(II) execute a file |
| exec, execl, execv(II) | execute a file |
| reset, setexit(III) | execute non-local goto |
| sleep(I) suspend | execution for an interval |
| sleep(II) stop | execution for interval |
| monitor(III) prepare | execution profile |
| profil(II) | execution time profile |
| exec, execl, | execv(II) execute a file |
| | exit(I) terminate command file |
| | exit(II) terminate process |
| | exp(III) exponential function |
| exp(III) | exponential function |
| pow(III) floating | exponentiation |
| gsi(VI) interpret | extended character set on GSI terminal |
| greek(V) graphics for | extended TTY-37 type-box |
| abs, | fabs(III) absolute value |
| factor(VI) discover prime | factors of a number |
| | factor(VI) discover prime factors of a number |
| abort(III) generate an IOT | fault |
| | fc(I) Fortran compiler |
| putc, putw, | fcreat, fflush(III) buffered output |
| ecvt, | fcvt(III) output conversion |
| | fed(VI) edit form letter memory |
| col(VI) filter reverse line | feeds |
| putc, putw, fcreat, | fflush(III) buffered output |
| diff(I) differential | file comparator |
| dup(II) duplicate an open | file descriptor |
| grep(I) search a | file for a pattern |
| ar(V) archive (library) | file format |
| split(I) split a | file into pieces |
| setfil(III) specify Fortran | file name |
| stat(II) get | file status |
| dcheck(VIII) | file system directory consistency check |
| dump(VIII) incremental | file system dump |
| restor(VIII) incremental | file system restore |
| icheck(VIII) | file system storage consistency check |
| mtab(V) mounted | file system table |
| fs(V) format of | file system volume |
| mkfs(VIII) construct a | file system |
| mount(II) mount | file system |
| mount(VIII) mount | file system |
| umount(II) dismount | file system |
| umount(VIII) dismount | file system |

-

| | |
|---|---|
| dir(V) | format of directories |
| fs(V) | format of file system volume |
| tbl(VI) | format tables for nroff or troff |
| nroff(I) | format text |
| roff(I) | format text |
| troff(I) | format text |
| ar(V) archive (library) file | format |
| dump(V) incremental dump tape | format |
| tp(V) DEC/mag tape | formats |
| printf(III) | formatted print |
| ms(VII) macros for | formatting manuscripts |
| | form(VI) form letter generator |
| fc(I) | Fortran compiler |
| ierror(III) catch | Fortran errors |
| setfil(III) specify | Fortran file name |
| iargc(III) get command arguments from | Fortran...getarg, |
| | fptrap(III) floating point interpreter |
| df(VIII) disk | free |
| alloc, | free(III) core allocator |
| read(II) read | from file |
| getarg, iargc(III) get command arguments | from Fortran |
| ncheck(VIII) generate names | from i-numbers |
| nlist(III) get entries | from name list |
| getpw(III) get name | from UID |
| | fstat(II) get status of open file |
| | fs(V) format of file system volume |
| atan, atan2(III) arc tangent | function |
| exp(III) exponential | function |
| fmod(III) floating modulo | function |
| gamma(III) log gamma | function |
| floor, ceil(III) floor and ceiling | functions |
| sqrt(III) square root | function |
| sin, cos(III) trigonometric | functions |
| bj(VI) the | game of black jack |
| chess(VI) the | game of chess |
| wump(VI) the | game of hunt-the-wumpus |
| ttt(VI) the | game of tic-tac-toe |
| moo(VI) guessing | game |
| gamma(III) log | gamma function |
| | gamma(III) log gamma function |
| m6(VI) | general purpose macroprocessor |
| tty(IV) | general typewriter interface |
| abort(III) | generate an IOT fault |
| glob(VIII) | generate command arguments |
| ncheck(VIII) | generate names from i-numbers |
| form(VI) form letter | generator |
| rand, srand(III) random number | generator |
| getarg, iargc(III) | get command arguments from Fortran |
| time(II) | get date and time |
| nlist(III) | get entries from name list |
| stat(II) | get file status |
| getgid(II) | get group identifications |
| getpw(III) | get name from UID |

-

| | |
|---|---|
| getpid(II) | get process identification |
| times(II) | get process times |
| fstat(II) | get status of open file |
| tty(I) | get typewriter name |
| gtty(II) | get typewriter status |
| getuid(II) | get user identifications |
| | getarg, iargc(III) get command arguments from Fortran |
| | getc, getw, fopen(III) buffered input |
| | getchar(III) read character |
| | getgid(II) get group identifications |
| | getpid(II) get process identification |
| | getpw(III) get name from UID |
| | getty(VIII) set typewriter mode |
| | getuid(II) get user identifications |
| getc, | getw, fopen(III) buffered input |
| | glob(VIII) generate command arguments |
| ctime, localtime, | gmtime(III) convert date and time to ASCII |
| | goto(I) command transfer |
| reset, setexit(III) execute non-local | goto |
| graph(VI) draw a | graph |
| plot: tek, gsip, vt0(VI) | graphics filters |
| greek(V) | graphics for extended TTY-37 type-box |
| plot: openpl et al.(VII) | graphics interface |
| | graph(VI) draw a graph |
| | greek(V) graphics for extended TTY-37 type-box |
| | grep(I) search a file for a pattern |
| group(V) | group file |
| getgid(II) get | group identifications |
| setgid(II) set process | group ID |
| chown(II) change owner and | group of a file |
| chgrp(VIII) change | group |
| newgrp(I) log in to a new | group |
| | group(V) group file |
| gsi(VI) interpret extended character set on | GSI terminal |
| plot: tek, | gsip, vt0(VI) graphics filters |
| | gsi(VI) interpret extended character set on GSI terminal |
| | gtty(II) get typewriter status |
| moo(VI) | guessing game |
| nohup(I) run a command immune to | hangups |
| hmul(III) | high-order product |
| wtmp(V) user login | history |
| | hmul(III) high-order product |
| | hp(IV) RH-11/RP04 moving-head disk |
| | hs(IV) RH11/RS03-RS04 fixed-head disk file |
| | ht(IV) RH-11/TU-16 magtape interface |
| wump(VI) the game of | hunt-the-wumpus |
| getarg, | iargc(III) get command arguments from Fortran |
| | icheck(VIII) file system storage consistency check |
| getpid(II) get process | identification |
| getgid(II) get group | identifications |
| getuid(II) get user | identifications |
| setgid(II) set process group | ID |
| setuid(II) set process user | ID |

-

| | |
|---|---|
| | ierror(III) catch Fortran errors |
| | if(I) conditional command |
| signal(II) catch or | ignore signals |
| core(V) format of core | image file |
| nohup(I) run a command | immune to hangups |
| uniq(I) report repeated lines | in a file |
| end, etext, edata(III) last locations | in program |
| newgrp(I) log | in to a new group |
| dump(V) | incremental dump tape format |
| dump(VIII) | incremental file system dump |
| restor(VIII) | incremental file system restore |
| indir(II) | indirect system call |
| | indir(II) indirect system call |
| utmp(V) user | information |
| ttys(V) typewriter | initialization data |
| init(VIII) process control | initialization |
| | init(VIII) process control initialization |
| clri(VIII) clear | i-node |
| getc, getw, fopen(III) buffered | input |
| atoi(III) convert ASCII to | integer |
| bc(I) arbitrary precision | interactive language |
| dsw(I) delete | interactively |
| cat(IV) phototypesetter | interface |
| dc(IV) DC-11 communications | interface |
| dn(IV) DN-11 ACU | interface |
| dp(IV) DP-11 201 data-phone | interface |
| ht(IV) RH-11/TU-16 magtape | interface |
| kl(IV) KL-11 or DL-11 asynchronous | interface |
| plot: openpl et al.(VII) graphics | interface |
| tm(IV) TM-11/TU-10 magtape | interface |
| tty(IV) general typewriter | interface |
| spline(VI) | interpolate smooth curve |
| gsi(VI) | interpret extended character set on GSI terminal |
| fptrap(III) floating point | interpreter |
| sh(I) shell (command | interpreter) |
| sno(VI) Snobol | interpreter |
| pipe(II) create an | interprocess channel |
| sleep(I) suspend execution for an | interval |
| sleep(II) stop execution for | interval |
| split(I) split a file | into pieces |
| intro(II) | introduction to system calls |
| | intro(II) introduction to system calls |
| ncheck(VIII) generate names from | i-numbers |
| abort(III) generate an | IOT fault |
| bj(VI) the game of black | jack |
| | kill(I) terminate a process |
| | kill(II) send signal to a process |
| kl(IV) | KL-11 or DL-11 asynchronous interface |
| | kl(IV) KL-11 or DL-11 asynchronous interface |
| mem, | kmem, null(IV) core memory |
| quiz(VI) test your | knowledge |
| bc(I) arbitrary precision interactive | language |
| primes(VI) print all primes | larger than somewhat |

-

| | |
|---|---|
| end, etext, edata(III) | last locations in program |
| | ld(I) link editor |
| | ldiv, lrem(III) long division |
| form(VI) form | letter generator |
| fed(VI) edit form | letter memory |
| ar(V) archive | (library) file format |
| ar(I) archive and | library maintainer |
| col(VI) filter reverse | line feeds |
| lpd(VIII) | line printer daemon |
| lp(IV) | line printer |
| opr(I) off | line print |
| comm(I) print | lines common to two files |
| uniq(I) report repeated | lines in a file |
| rev(I) reverse | lines of a file |
| a.out(V) assembler and | link editor output |
| ld(I) | link editor |
| link(II) | link to a file |
| | link(II) link to a file |
| ln(I) make a | link |
| ls(I) | list contents of directory |
| cref(I) make cross reference | listing |
| nlist(III) get entries from name | list |
| nm(I) print name | list |
| | ln(I) make a link |
| ctime, | localtime, gmtime(III) convert date and time to ASCII |
| end, etext, edata(III) last | locations in program |
| | locv(III) long output conversion |
| gamma(III) | log gamma function |
| newgrp(I) | log in to a new group |
| log(III) natural | logarithm |
| | log(III) natural logarithm |
| ac(VIII) | login accounting |
| wtmp(V) user | login history |
| passwd(I) change | login password |
| | login(I) sign onto UNIX |
| ldiv, lrem(III) | long division |
| locv(III) | long output conversion |
| nice(I) run a command at | low priority |
| | lpd(VIII) line printer daemon |
| | lp(IV) line printer |
| ldiv, | lrem(III) long division |
| | ls(I) list contents of directory |
| | m6(VI) general purpose macroprocessor |
| m6(VI) general purpose | macroprocessor |
| ms(VII) | macros for formatting manuscripts |
| ht(IV) RH-11/TU-16 | magtape interface |
| tm(IV) TM-11/TU-10 | magtape interface |
| tp(I) manipulate DECtape and | magtape |
| mail(I) send | mail to designated users |
| | mail(I) send mail to designated users |
| ar(I) archive and library | maintainer |
| mknod(II) | make a directory or a special file |
| mkdir(I) | make a directory |

-

| | |
|---|---|
| ln(I) | make a link |
| cref(I) | make cross reference listing |
| | man(I) run off section of UNIX manual |
| tp(I) | manipulate DECtape and magtape |
| salloc(VII) string allocation and | manipulation |
| man(I) run off section of UNIX | manual |
| ms(VII) macros for formatting | manuscripts |
| ascii(V) | map of ASCII character set |
| neqn(I) typeset | mathematics on terminal |
| eqn(I) typeset | mathematics |
| | mem, kmem, null(IV) core memory |
| fed(VI) edit form letter | memory |
| mem, kmem, null(IV) core | memory |
| sort, usort(I) sort or | merge files |
| | mesg(I) permit or deny messages |
| mesg(I) permit or deny | messages |
| sysnerr, errno(III) system error | messages...perror, syserrlist, |
| | mkdir(I) make a directory |
| | mkfs(VIII) construct a file system |
| | mknod(II) make a directory or a special file |
| | mknod(VIII) build special file |
| chmod(II) change | mode of file |
| stty(II) set | mode of typewriter |
| chmod(I) change | mode |
| getty(VIII) set typewriter | mode |
| fmod(III) floating | modulo function |
| | monitor(III) prepare execution profile |
| | moo(VI) guessing game |
| mount(II) | mount file system |
| mount(VIII) | mount file system |
| mtab(V) | mounted file system table |
| | mount(II) mount file system |
| | mount(VIII) mount file system |
| mv(I) | move or rename a file |
| seek(II) | move read/write pointer |
| hp(IV) RH-11/RP04 | moving-head disk |
| rp(IV) RP-11/RP03 | moving-head disk |
| | ms(VII) macros for formatting manuscripts |
| | mtab(V) mounted file system table |
| dh(IV) DH-11 communications | multiplexer |
| | mv(I) move or rename a file |
| getpw(III) get | name from UID |
| nlist(III) get entries from | name list |
| nm(I) print | name list |
| ttyn(III) return | name of current typewriter |
| pwd(I) working directory | name |
| ncheck(VIII) generate | names from i-numbers |
| setfil(III) specify Fortran file | name |
| tty(I) get typewriter | name |
| | nargs(III) argument count |
| log(III) | natural logarithm |
| | ncheck(VIII) generate names from i-numbers |
| | neqn(I) typeset mathematics on terminal |

-

| | |
|---:|:---|
| creat(II) create a | new file |
| newgrp(I) log in to a | new group |
| fork(II) spawn | new process |
| | newgrp(I) log in to a new group |
| | nice(I) run a command at low priority |
| | nice(II) set program priority |
| | nlist(III) get entries from name list |
| | nm(I) print name list |
| | nohup(I) run a command immune to hangups |
| reset, setexit(III) execute | non-local goto |
| tbl(VI) format tables for | nroff or troff |
| | nroff(I) format text |
| mem, kmem, | null(IV) core memory |
| rand, srand(III) random | number generator |
| factor(VI) discover prime factors of a | number |
| size(I) size of an | object file |
| sky(VI) | obtain ephemerides |
| od(I) | octal dump |
| | od(I) octal dump |
| opr(I) | off line print |
| man(I) run | off section of UNIX manual |
| login(I) sign | onto UNIX |
| dup(II) duplicate an | open file descriptor |
| fstat(II) get status of | open file |
| open(II) | open for reading or writing |
| | open(II) open for reading or writing |
| plot: | openpl et al.(VII) graphics interface |
| | opr(I) off line print |
| stty(I) set typewriter | options |
| rk(IV) RK-11/RK03 | (or RK05) disk |
| ecvt, fcvt(III) | output conversion |
| locv(III) long | output conversion |
| a.out(V) assembler and link editor | output |
| putc, putw, fcreat, fflush(III) buffered | output |
| chown(II) change | owner and group of a file |
| chown(VIII) change | owner |
| pc(IV) PC-11 | paper tape reader/punch |
| | passwd(I) change login password |
| | passwd(V) password file |
| crypt(III) | password encoding |
| passwd(V) | password file |
| passwd(I) change login | password |
| grep(I) search a file for a | pattern |
| pc(IV) | PC-11 paper tape reader/punch |
| | pc(IV) PC-11 paper tape reader/punch |
| update(VIII) | periodically update the super block |
| mesg(I) | permit or deny messages |
| error messages... | perror, syserrlist, sysnerr, errno(III) system |
| | pfe(I) print floating exception |
| dpd(VIII) data | phone daemon |
| cat(IV) | phototypesetter interface |
| split(I) split a file into | pieces |
| tee(I) | pipe fitting |

| | |
|---|---|
| | pipe(II) create an interprocess channel |
| | plot: openpl et al.(VII) graphics interface |
| | plot: tek, gsip, vt0(VI) graphics filters |
| fptrap(III) floating | point interpreter |
| seek(II) move read/write | pointer |
| typo(I) find | possible typos |
| | pow(III) floating exponentiation |
| bc(I) arbitrary | precision interactive language |
| azel(VI) satellite | predictions |
| monitor(III) | prepare execution profile |
| | pr(I) print file |
| factor(VI) discover | prime factors of a number |
| primes(VI) print all | primes larger than somewhat |
| | primes(VI) print all primes larger than somewhat |
| primes(VI) | print all primes larger than somewhat |
| date(I) | print and set the date |
| cal(VI) | print calendar |
| pr(I) | print file |
| pfe(I) | print floating exception |
| comm(I) | print lines common to two files |
| nm(I) | print name list |
| cat(I) concatenate and | print |
| lpd(VIII) line | printer daemon |
| lp(IV) line | printer |
| | printf(III) formatted print |
| opr(I) off line | print |
| printf(III) formatted | print |
| nice(I) run a command at low | priority |
| nice(II) set program | priority |
| su(VIII) become | privileged user |
| boot | procedures(VIII) UNIX startup |
| init(VIII) | process control initialization |
| setgid(II) set | process group ID |
| getpid(II) get | process identification |
| ps(I) | process status |
| times(II) get | process times |
| wait(II) wait for | process to terminate |
| ptrace(II) | process trace |
| setuid(II) set | process user ID |
| exit(II) terminate | process |
| fork(II) spawn new | process |
| kill(I) terminate a | process |
| kill(II) send signal to a | process |
| wait(I) await completion of | process |
| hmul(III) high-order | product |
| | prof(I) display profile data |
| prof(I) display | profile data |
| monitor(III) prepare execution | profile |
| profil(II) execution time | profile |
| | profil(II) execution time profile |
| nice(II) set | program priority |
| end, etext, edata(III) last locations in | program |
| units(VI) conversion | program |

-

| | |
|---|---|
| | ps(I) process status |
| | ptrace(II) process trace |
| m6(VI) general | purpose macroprocessor |
| | putc, putw, fcreat, fflush(III) buffered output |
| | putchar, flush(III) write character |
| putc, | putw, fcreat, fflush(III) buffered output |
| | pwd(I) working directory name |
| | qsort(III) quicker sort |
| qsort(III) | quicker sort |
| | quiz(VI) test your knowledge |
| | rand, srand(III) random number generator |
| rand, srand(III) | random number generator |
| rc(I) | Ratfor compiler |
| | rc(I) Ratfor compiler |
| getchar(III) | read character |
| csw(II) | read console switches |
| read(II) | read from file |
| pc(IV) PC-11 paper tape | reader/punch |
| | read(II) read from file |
| open(II) open for | reading or writing |
| seek(II) move | read/write pointer |
| cref(I) make cross | reference listing |
| strip(I) remove symbols and | relocation bits |
| unlink(II) | remove directory entry |
| rmdir(I) | remove directory |
| strip(I) | remove symbols and relocation bits |
| rm(I) | remove (unlink) files |
| mv(I) move or | rename a file |
| uniq(I) report | repeated lines in a file |
| uniq(I) | report repeated lines in a file |
| | reset, setexit(III) execute non-local goto |
| restor(VIII) incremental file system | restore |
| | restor(VIII) incremental file system restore |
| ttyn(III) | return name of current typewriter |
| col(VI) filter | reverse line feeds |
| rev(I) | reverse lines of a file |
| | rev(I) reverse lines of a file |
| rf(IV) | RF11/RS11 fixed-head disk file |
| | rf(IV) RF11/RS11 fixed-head disk file |
| hp(IV) | RH-11/RP04 moving-head disk |
| hs(IV) | RH11/RS03-RS04 fixed-head disk file |
| ht(IV) | RH-11/TU-16 magtape interface |
| rk(IV) RK-11/RK03 (or | RK05) disk |
| rk(IV) | RK-11/RK03 (or RK05) disk |
| | rk(IV) RK-11/RK03 (or RK05) disk |
| | rmdir(I) remove directory |
| | rm(I) remove (unlink) files |
| | roff(I) format text |
| sqrt(III) square | root function |
| rp(IV) | RP-11/RP03 moving-head disk |
| | rp(IV) RP-11/RP03 moving-head disk |
| nice(I) | run a command at low priority |
| nohup(I) | run a command immune to hangups |

| | |
|---:|:---|
| man(I) | run off section of UNIX manual |
| | salloc(VII) string allocation and manipulation |
| azel(VI) | satellite predictions |
| | sa(VIII) Shell accounting |
| break, brk, | sbrk(II) change core allocation |
| crwrite, crexch, crprior(VII) coroutine | scheme...crfork, crexit, crread, |
| grep(I) | search a file for a pattern |
| man(I) run off | section of UNIX manual |
| | seek(II) move read/write pointer |
| mail(I) | send mail to designated users |
| kill(II) | send signal to a process |
| stty(II) | set mode of typewriter |
| gsi(VI) interpret extended character | set on GSI terminal |
| setgid(II) | set process group ID |
| setuid(II) | set process user ID |
| nice(II) | set program priority |
| tabs(V) | set tab stops |
| date(I) print and | set the date |
| stime(II) | set time |
| getty(VIII) | set typewriter mode |
| stty(I) | set typewriter options |
| ascii(V) map of ASCII character | set |
| reset, | setexit(III) execute non-local goto |
| | setfil(III) specify Fortran file name |
| | setgid(II) set process group ID |
| | setuid(II) set process user ID |
| sa(VIII) | Shell accounting |
| shift(I) adjust | Shell arguments |
| sh(I) | shell (command interpreter) |
| | sh(I) shell (command interpreter) |
| | shift(I) adjust Shell arguments |
| login(I) | sign onto UNIX |
| kill(II) send | signal to a process |
| | signal(II) catch or ignore signals |
| signal(II) catch or ignore | signals |
| | sin, cos(III) trigonometric functions |
| size(I) | size of an object file |
| | size(I) size of an object file |
| | sky(VI) obtain ephemerides |
| | sleep(I) suspend execution for an interval |
| | sleep(II) stop execution for interval |
| spline(VI) interpolate | smooth curve |
| sno(VI) | Snobol interpreter |
| | sno(VI) Snobol interpreter |
| primes(VI) print all primes larger than | somewhat |
| sort, usort(I) | sort or merge files |
| | sort, usort(I) sort or merge files |
| qsort(III) quicker | sort |
| fork(II) | spawn new process |
| | speak(VI) word to voice translator |
| mknod(II) make a directory or a | special file |
| mknod(VIII) build | special file |
| setfil(III) | specify Fortran file name |

-

| | |
|---|---|
| | tabs(V) set tab stops |
| atan, atan2(III) arc | tangent function |
| dump(V) incremental dump | tape format |
| tp(V) DEC/mag | tape formats |
| pc(IV) PC-11 paper | tape reader/punch |
| | tbl(VI) format tables for nroff or troff |
| tc(IV) | TC-11/TU56 DECtape |
| | tc(IV) TC-11/TU56 DECtape |
| | tee(I) pipe fitting |
| plot: | tek, gsip, vt0(VI) graphics filters |
| interpret extended character set on GSI | terminal...gsi(VI) |
| neqn(I) typeset mathematics on | terminal |
| kill(I) | terminate a process |
| exit(I) | terminate command file |
| exit(II) | terminate process |
| wait(II) wait for process to | terminate |
| quiz(VI) | test your knowledge |
| ed(I) | text editor |
| nroff(I) format | text |
| roff(I) format | text |
| troff(I) format | text |
| primes(VI) print all primes larger | than somewhat |
| cubic(VI) | three dimensional tic-tac-toe |
| cubic(VI) three dimensional | tic-tac-toe |
| ttt(VI) the game of | tic-tac-toe |
| time(I) | time a command |
| profil(II) execution | time profile |
| localtime, gmtime(III) convert date and | time to ASCII...ctime, |
| | time(I) time a command |
| | time(II) get date and time |
| | times(II) get process times |
| stime(II) set | time |
| times(II) get process | times |
| time(II) get date and | time |
| tm(IV) | TM-11/TU-10 magtape interface |
| | tmg(VI) compiler-compiler |
| | tm(IV) TM-11/TU-10 magtape interface |
| | tp(I) manipulate DECtape and magtape |
| | tp(V) DEC/mag tape formats |
| ptrace(II) process | trace |
| goto(I) command | transfer |
| speak(VI) word to voice | translator |
| tr(I) | transliterate |
| | tr(I) transliterate |
| sin, cos(III) | trigonometric functions |
| | troff(I) format text |
| tbl(VI) format tables for nroff or | troff |
| | ttt(VI) the game of tic-tac-toe |
| greek(V) graphics for extended | TTY-37 type-box |
| | tty(I) get typewriter name |
| | tty(IV) general typewriter interface |
| | ttyn(III) return name of current typewriter |
| | ttys(V) typewriter initialization data |

-

| | |
|---|---|
| cmp(I) compare | two files |
| comm(I) print lines common to | two files |
| greek(V) graphics for extended TTY-37 | type-box |
| file(I) determine file | type |
| neqn(I) | typeset mathematics on terminal |
| eqn(I) | typeset mathematics |
| ttys(V) | typewriter initialization data |
| tty(IV) general | typewriter interface |
| getty(VIII) set | typewriter mode |
| tty(I) get | typewriter name |
| stty(I) set | typewriter options |
| gtty(II) get | typewriter status |
| stty(II) set mode of | typewriter |
| ttyn(III) return name of current | typewriter |
| | typo(I) find possible typos |
| typo(I) find possible | typos |
| getpw(III) get name from | UID |
| | umount(II) dismount file system |
| | umount(VIII) dismount file system |
| | uniq(I) report repeated lines in a file |
| | units(VI) conversion program |
| man(I) run off section of | UNIX manual |
| boot procedures(VIII) | UNIX startup |
| login(I) sign onto | UNIX |
| rm(I) remove | (unlink) files |
| | unlink(II) remove directory entry |
| sync(II) | update super-block |
| sync(VIII) | update the super block |
| update(VIII) periodically | update the super block |
| | update(VIII) periodically update the super block |
| du(I) summarize disk | usage |
| getuid(II) get | user identifications |
| setuid(II) set process | user ID |
| utmp(V) | user information |
| wtmp(V) | user login history |
| mail(I) send mail to designated | users |
| su(VIII) become privileged | user |
| wall(VIII) write to all | users |
| write(I) write to another | user |
| sort, | usort(I) sort or merge files |
| | utmp(V) user information |
| abs, fabs(III) absolute | value |
| speak(VI) word to | voice translator |
| fs(V) format of file system | volume |
| plot: tek, gsip, | vt0(VI) graphics filters |
| wait(II) | wait for process to terminate |
| | wait(I) await completion of process |
| | wait(II) wait for process to terminate |
| | wall(VIII) write to all users |
| | wc(I) word count |
| crash(VIII) | what to do when the system crashes |
| crash(VIII) what to do | when the system crashes |
| who(I) | who is on the system |

-

| | |
|---|---|
| | who(I) who is on the system |
| wc(I) | word count |
| speak(VI) | word to voice translator |
| pwd(I) | working directory name |
| chdir(I) change | working directory |
| chdir(II) change | working directory |
| putchar, flush(III) | write character |
| write(II) | write on a file |
| wall(VIII) | write to all users |
| write(I) | write to another user |
| | write(I) write to another user |
| | write(II) write on a file |
| open(II) open for reading or | writing |
| | wtmp(V) user login history |
| | wump(VI) the game of hunt-the-wumpus |
| | yacc(I) yet another compiler-compiler |
| yacc(I) | yet another compiler-compiler |
| quiz(VI) test | your knowledge |

## NAME

ar – archive and library maintainer

## SYNOPSIS

**ar** key afile name ...

## DESCRIPTION

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

*Key* is one character from the set **drtux,** optionally concatenated with **v.** *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 archive file does not exist, **r** creates it. If the named files are not in the archive file, they are appended.

**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.

**u** is similar to **r** except that only those files that have been modified are replaced. If no names are given, all files in the archive that have been modified are replaced by the modified version.

**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. The following abbreviations are used:

| | |
|---|---|
| **c** | copy |
| **a** | append |
| **d** | delete |
| **r** | replace |
| **x** | extract |

## FILES

| | |
|---|---|
| /tmp/vtm? | temporary |

## SEE ALSO

ld (I), archive (V)

## BUGS

Option **tv** should be implemented as a table with more information.

There should be a way to specify the placement of a new file in an archive. Currently, it is placed at the end.

Since *ar* has not been rewritten to deal properly with the new file system modes, extracted files have mode 666.

For the same reason, only the first 8 characters of file names are significant.

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

## NAME

as − assembler

## SYNOPSIS

**as** [ − ] 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 **a.out.** 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 Manual'.

## 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

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.

ˆ
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* (I), C Reference Manual, ‘‘BC − An Arbitrary Precision Desk-Calculator Language.’’

**BUGS**

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

-

**NAME**

cat – concatenate and print

**SYNOPSIS**

**cat** 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.

**SEE ALSO**

pr(I), cp(I)

**DIAGNOSTICS**

none; if a file cannot be found it is ignored.

**BUGS**

**cat x y >x** and **cat x y >y** cause strange results.

## NAME

cc – C compiler

## SYNOPSIS

**cc** [ **–c** ] [ **–p** ] [ **–f** ] [ **–O** ] [ **–S** ] [ **–P** ] file ...

## 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* subroutine (III) 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.

**–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'.

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.**

## 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/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

''Programming in C— a tutorial,'' C Reference Manual, monitor (III), prof (I), cdb (I), ld (I).

## DIAGNOSTICS

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

-

**BUGS**

## 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’’) and 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 typeout 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) 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

## 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

## 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.

## SEE ALSO

sh (I), pwd (I)

## BUGS

**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’’ option 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)

**BUGS**

## 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 (I), comm (I)

## BUGS

-

## NAME

comm – print lines common to two files

## SYNOPSIS

**comm** [ – [ **123** ] ] file1 file2

## DESCRIPTION

*Comm* reads *file1* and *file2*, which should be in sort, 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)

## BUGS

## 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

cat (I), pr (I), mv (I)

## BUGS

**NAME**

cref – make cross reference listing

**SYNOPSIS**

**cref** [ **–acilostux123** ] 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) |
| **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/cign | default C *ignore* file |
| /usr/bin/crpost | post processor |
| /usr/bin/upost | post processor for –**u** option |

/bin/sort used to sort temporaries

**SEE ALSO**

as (I), cc (I)

**BUGS**

**NAME**

date – print and set the date

**SYNOPSIS**

**date** [ **s** ] [ mmddhhmm[yy] ]

**DESCRIPTION**

If no argument is given, the current date and time are printed. If an argument is given, 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 is ‘‘s,’’ *date* calls the network file store via the TIU interface (if present) and sets the clock to the time thereby obtained.

**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.

**FILES**

/dev/tiu/d0

**BUGS**

## NAME

db – debug

## SYNOPSIS

**db** [ core [ namelist ] ] [ – ]

## DESCRIPTION

Unlike many debugging packages (including DEC'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.

ˆ 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 ˆ 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.

$ 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.

+ – * % ˆ
: The top two values on the stack are added (+), subtracted (–), multiplied (*), divided (/), remaindered (%), or exponentiated (ˆ). 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.

**BUGS**

## 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 comma-separated conversions |

Where sizes are specified, a number of bytes is expected. A number may end with **k, b** or **w** to specify multiplication by 1024, 512, or 2 respectively. Also 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. There should be separate options.

**NAME**

diff – differential file comparator

**SYNOPSIS**

**diff** [ – ] name1 name2

**DESCRIPTION**

*Diff* tells what lines must be changed in two files to bring them into agreement. 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 '**.**'.

Under the – option, the output of *diff* is a script of *a, c* and *d* commands for the editor *ed,* which will change the contents of the first file into the contents of the second. In this connection, 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.

**SEE ALSO**

cmp (I), comm (I), ed (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 – option are naive about creating lines consisting of a single '**.**'.

**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.

**BUGS**

**NAME**

ed – text editor

**SYNOPSIS**

**ed** [ – ] [ name ]

**DESCRIPTION**

*Ed* is the standard text editor.

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

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

Commands to *ed* have a simple and regular structure: zero or more *addresses* followed by a single character *command,* possibly followed by parameters to the command. These addresses specify one or more lines in the buffer. Every command which 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. 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:

1. An ordinary character (not one of those discussed below) is a regular expression and matches that character.
2. A circumflex '^' at the beginning of a regular expression matches the empty string at the beginning of a line.
3. A currency symbol '$' at the end of a regular expression matches the null character at the end of a line.
4. A period '**.**' matches any character except a new-line character.
5. A regular expression followed by an asterisk '*' matches any number of adjacent occurrences (including zero) of the regular expression it follows.
6. A string of characters enclosed in square brackets '[ ]' matches any character in the string but no others. If, however, the first character of the string is a circumflex '^' the regular expression matches any character except new-line and the characters in the string.
7. The concatenation of regular expressions is a regular expression which matches the concatenation of the strings matched by the components of the regular expression.
8. A regular expression enclosed between the sequences '\(' and '\)'is identical to the unadorned expression; the construction has side effects discussed under the *s* command.
9. The null regular expression standing alone is equivalent to the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see *s* below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by '\'. This also applies to the character bounding the regular expression (often '/') and to '\' itself.

To understand addressing in *ed* it is necessary to know that at any time there is a *current line.* Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character '**.**' addresses the current line.
2. The character '$' addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*-th line of the buffer.
4. '´*x*' addresses the line marked with the 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 in slashes '/' addresses the first line found by searching 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.
6. A regular expression enclosed in queries '?' addresses the first line found by searching 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.
7. An address followed by a plus sign '+' or a minus sign '−' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with '+' or '−' the addition or subtraction is taken with respect to the current line; e.g. '−5' is understood to mean '**.**−5'.
9. If an address ends with '+' or '−', then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address '−' refers to the line before the current line. Moreover, trailing '+' and '−' characters have cumulative effect, so '−−' refers to the current line less 2.
10. To maintain compatibility with earlier version of the editor, the character '^' in addresses is entirely equivalent to '−'.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma '**,**'. They may also be separated by a semicolon '**;**'. In this case the current line '**.**' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches ('/', '?'). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of *ed* commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, any command may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed respectively in the way discussed below.

( **.** ) a
<text>
**.**

The append command reads the given text and appends it after the addressed line. '**.**' is left on the last line input, if there were any, otherwise at the addressed line. Address '0' is legal for this command; text is placed at the beginning of the buffer.

( **.** , **.** ) c
<text>
**.**

The change command deletes the addressed lines, then accepts input text which replaces these lines. '**.**' is left at the last line input; if there were none, it is left at the first line not deleted.

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

e filename
The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. '**.**' is set to the last line of the buffer. The number of characters read is typed. 'filename' is remembered for possible use as a default file name in a subsequent *r* or *w* command.

f filename
The filename command prints the currently remembered file name. If 'filename' is given, the currently remembered file name is changed to 'filename'.

(1,$) g/regular expression/command list
In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with '**.**' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with '\'. *A, i,* and *c* commands and associated input are permitted; the '**.**' terminating input mode may be omitted if it would be on the last line of the command list. The (global) commands, *g,* and *v,* are not permitted in the command list.

( **.** ) i
<text>
**.**
This command inserts the given text before the addressed line. '**.**' is left at the last line input; if there were none, at the addressed line. This command differs from the *a* command only in the placement of the text.

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

( **.** , **.** ) l
The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in octal, and long lines are folded. An *l* command may follow any other on the same line.

( **.** , **.** ) m*a*
The move command repositions the addressed lines after the line addressed by *a*. The last of the moved lines becomes the current line.

( **.** , **.** ) p
The print command prints the addressed lines. '**.**' is left at the last line printed. The *p* command may be placed on the same line after any command.

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

($) r filename
The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see *e* and *f* commands). The remembered file name is not changed unless 'filename' is the very first file name mentioned. Address '0' is legal for *r* and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. '**.**' is left at the last line read in from the file.

( **.** , **.** ) s/regular expression/replacement/ or,
( **.** , **.** ) s/regular expression/replacement/g
The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are re-

placed by the replacement specified, if the global replacement indicator ‘g’ appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of ‘/’ to delimit the regular expression and the replacement. ‘**.**’ is left at the last line substituted.

An ampersand ‘&’ appearing in the replacement is replaced by the string matching the regular expression. The special meaning of ‘&’ in this context may be suppressed by preceding it by ‘\’. 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 enclosed between ‘\(’ and ‘\)’. When nested, parenthesized subexpressions are present, *n* is determined by counting occurrences of ‘\(’ starting from the left.

Lines may be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by ‘\’.

( **.** , **.** ) 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 on the last line of the copy.

(1,$) v/regular expression/command list
This command is the same as the global command except that the command list is executed with ‘**.**’ initially set to every line *except* those matching the regular expression.

(1,$) w filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writeable by everyone). The remembered file name is *not* changed unless ‘filename’ is the very first file name mentioned. 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 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 UNIX to be interpreted as a command. ‘**.**’ is unchanged.

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

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

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

**FILES**

/tmp/#, temporary; ‘#’ is the process number (in octal).

**DIAGNOSTICS**

‘?’ for errors in commands; ‘TMP’ for temporary file overflow.

**SEE ALSO**

A Tutorial Introduction to the ED Text Editor (B. W. Kernighan)

**BUGS**

The *s* command causes all marks to be lost on lines changed.

## NAME

eqn – typeset mathematics

## SYNOPSIS

**eqn** [ file ] ...

## DESCRIPTION

*Eqn* is a troff (I) preprocessor for typesetting mathematics on the Graphics Systems 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 to get centering, numbering, etc. All other lines are treated as comments, and passed through untouched.

Spaces, tabs, newlines, 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 $\begin{matrix} a \\ b \\ c \end{matrix}$. There can be an arbitrary number of elements in a pile. **lpile** left-justifies, **pile** and **cpile** center, with different vertical spacing, and **rpile** right justifies.

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

Keywords like *sum* ( $\sum$ ) *int* ( $\int$ ) *inf* ( $\infty$ ) and shorthands like >= ( $\geq$ ) –> ( $\rightarrow$ ), != ( $\neq$ ), 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 \(bs (\) can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched.

## SEE ALSO

A System for Typesetting Mathematics (Computer Science Technical Report #17, Bell Laboratories, 1974.)
TROFF Users’ Manual (internal memorandum)
TROFF Made Trivial (internal memorandum)
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**

exit – terminate command file

**SYNOPSIS**

**exit**

**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.

**SEE ALSO**

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

**BUGS**

**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 units; 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 identical to the C preprocessor is called, which implements ''#define'' and ''#include'' 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/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 more 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

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.

## BUGS

**NAME**

find – find files

**SYNOPSIS**

**find** pathname expression

**DESCRIPTION**

*Find* recursively descends the directory hierarchy from *pathname* 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 exit status zero (most commands do). The end of the command is 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 files named 'a.out' and '*.o' not 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), file system (V)

**BUGS**

There is no way to check device type.
Syntax should be reconciled with *if.*

## 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)

## BUGS

## NAME

grep – search a file for a pattern

## SYNOPSIS

**grep** [ –**v** ] [ –**b** ] [ –**c** ] [ –**n** ] 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 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). Care should be taken when using the characters $ * [ ˆ | ( ) 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)

## BUGS

Lines are limited to 256 characters; longer lines are truncated.

-

## NAME

if – conditional command

## SYNOPSIS

**if** expr command [ arg ... ]

## DESCRIPTION

*If* evaluates the expression *expr*, and if its value is true, executes the given *command* with the given arguments.

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

## SEE ALSO

sh (I), find (I)

## BUGS

-

**NAME**

kill – terminate a process

**SYNOPSIS**

**kill** [ –signo ] processid ...

**DESCRIPTION**

Kills the specified processes. The process number of each asynchronous process started with ‘&’ is reported by the Shell. Process numbers can also be found by using *ps* (I).

If process number 0 is used, then all processes belonging to the current user and associated with the same control typewriter are killed.

The killed process must belong to the current user unless he is the super-user.

If a signal number preceded by ‘‘–’’ is given as first argument, that signal is sent instead of *kill* (see *signal (II)).*

**SEE ALSO**

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

**BUGS**

**NAME**

ld – link editor

**SYNOPSIS**

**ld** [ **−sulxrdni** ] name ...

**DESCRIPTION**

*Ld* combines several object programs into one; resolves external references; and searches libraries. In the simplest case the names of several object programs 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 beginning of the first routine.

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** 'squash' 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.*

**−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* is any character. 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 the 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.

**FILES**

| | |
|---|---|
| /lib/lib?.a | libraries |
| a.out | output file |

**SEE ALSO**

as (I), ar (I)

-

**BUGS**

**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 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)

## DIAGNOSTICS

‘login incorrect,’ if the name or the password is bad. ‘No Shell,’, ‘cannot open password file,’ ‘no directory’: consult a UNIX programming counselor.

## BUGS

**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**.

-

**BUGS**

-

**NAME**

mail – send mail to designated users

**SYNOPSIS**

**mail** [ **−yn** ] [ person ... ]

**DESCRIPTION**

*Mail* with no argument searches for a file called prints it if it is nonempty, then asks if it should be saved. If the answer is **y,** the mail is added to *mbox.* Finally is truncated to zero length. To leave the mailbox untouched, hit ‘delete.’ The question can be answered on the command line with the argument ‘−y’ or ‘−n’.

When *persons* are named, *mail* takes the standard input up to an end of file and adds it to each *person's* file. The message is preceded by the sender's name and a postmark.

A *person* is either a user name recognized by *login* (I), in which case the mail is sent to the default working directory of that user; or the path name of a directory, in which case in that directory is used.

When a user logs in he is informed of the presence of mail. No mail will be received from a sender to whom is inaccessible or unwritable.

**FILES**

| | |
|---|---|
| /etc/passwd | to identify sender and locate persons |
| /etc/utmp | to identify sender |
| .mail | input mail |
| mbox | saved mail |
| /tmp/m# temp file | |

**SEE ALSO**

write (I)

**BUGS**

-

**NAME**

man – run off section of UNIX manual

**SYNOPSIS**

**man** [ section ] [ title ... ]

**DESCRIPTION**

*Man* is a shell command file which locates and prints one or more sections of this manual. *Section* is the section number of the manual, as an Arabic not Roman numeral, and is optional. *Title* is one or more section names; these names bear a generally simple relation to the page captions in the manual. If the *section* is missing, **1** is assumed. For example,

**man man**

would reproduce this page.

**FILES**

/usr/man/man?/*

**BUGS**

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

## NAME

mesg – permit or deny messages

## SYNOPSIS

**mesg** [ **n** ] [ **y** ]

## DESCRIPTION

*Mesg* with argument **n** forbids messages via *write* by revoking non-user write permission on the user's typewriter. *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 typewriter

## BUGS

-

## NAME

mkdir – make a directory

## SYNOPSIS

**mkdir** dirname ...

## DESCRIPTION

*Mkdir* creates specified directories in mode 777. The standard entries '**.**' and '**..**' are made automatically.

## SEE ALSO

rmdir (I)

## BUGS

## 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.

## 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 terminal.

If no arguments are specified, *neqn* reads the standard input, so it may be used as a filter.

## SEE ALSO

eqn (I), gsi (VI)

## BUGS

Because of some interactions with *nroff* 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*. 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)

**BUGS**

**NAME**

nice – run a command at low priority

**SYNOPSIS**

**nice** [ −*number* ] command [ arguments ]

**DESCRIPTION**

*Nice* executes *command* with low scheduling priority. If a numerical 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)

**BUGS**

-

## NAME

nm – print name list

## SYNOPSIS

**nm** [ **−cnrupg** ] [ name ]

## DESCRIPTION

*Nm* prints the symbol table from the output file of an assembler 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

## BUGS

-

**NAME**

nohup – run a command immune to hangups

**SYNOPSIS**

**nohup** command [ arguments ]

**DESCRIPTION**

*Nohup* executes *command* with hangups, quits and interrupts all ignored.

**SEE ALSO**

nice (I), signal (II)

**BUGS**

-

## NAME

nroff – format text

## SYNOPSIS

**nroff** [ **+***n* ] [ –*n* ] [ **−n***n* ] [ **−r***an* ] [ **−m***x* ] [ –**s** ] [ –**h** ] [ –**q** ] files

## DESCRIPTION

*Nroff* formats text according to control lines embedded in the text files. *Nroff* reads the standard input if no file arguments are given. An argument of just ''**−**'' refers to the standard input. The non-file option arguments are interpreted as follows:

| | |
|---|---|
| **+***n* | Output commences at the first page whose page number is *n* or larger. |
| –*n* | Printing stops after page *n.* |
| **−n***n* | First generated (not necessarily printed) page is given number *n*; simulates ''.pn *n*''. |
| **−r***an* | Set number register to the value *n.* |
| **−m***name* | Prepends a standard macro file; simulates ''.so /usr/lib/tmac.*name*''. |
| –**s** | Stop prior to each page to permit paper loading. Printing is restarted by typing a 'newline' character. |
| –**h** | Spaces are replaced where possible with tabs to speed up output (or reduce the size of the output file). |
| –**q** | Prompt names for insertions are not printed and the bell character is sent instead; the insertion is not echoed. |

## FILES

| | |
|---|---|
| /usr/lib/suftab | suffix hyphenation tables |
| /tmp/rtm? | temporary |
| /usr/lib/tmac.* | standard macro files |

## SEE ALSO

NROFF User's Manual (internal memorandum).
neqn (I), col (I)

## BUGS

-

## 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)

## BUGS

**NAME**

opr – off line print

**SYNOPSIS**

**opr** [ –destination ] [ –**crm** ] [ name ... ]

**DESCRIPTION**

*Opr* causes the named files to be printed off line at the specified destination. If no names appear the standard input is assumed.

At the mother system the following destinations are recognized. The default destination is **mh.**

**lp** Local line printer.

**mh** GCOS at Murray Hill Comp Center. GCOS identification must be registered in the UNIX password file (see passwd (V)).

**sp** Spider network printer.

*xx* The two-character code *xx* is taken to be a Murray Hill GCOS station id. Useful codes are ‘r1’ for quality print and ‘q1’ for quality print with special ribbon.

*Opr* uses spooling daemons that do the job when facilities become available. Flag –**r** causes the named files to be removed when spooled. Flag –**c** causes copies to be made so as to insulate the daemons from any intervening changes to the files.

Flag –**m** causes mail to be sent when UNIX is finished transmitting the file. For GCOS jobs the mail includes the snumb.

**FILES**

| | |
|---|---|
| /etc/passwd | personal ident cards |
| /lib/dpr | dataphone spooler |
| /etc/dpd | dataphone daemon |
| /usr/dpd/* | spool area |
| /lib/lpr | line printer spooler |
| /etc/lpd | line printer daemon |
| /usr/lpd/* | spool area |
| /lib/npr | spider network spooler |

**SEE ALSO**

fsend (I), dpd (VIII), lpd (VIII)

**BUGS**

Line printer spooler doesn’t handle flags.
Spider network spooler doesn’t spool.

**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 corresponding user 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)

**BUGS**

**NAME**

pfe – print floating exception

**SYNOPSIS**

**pfe**

**DESCRIPTION**

*Pfe* examines the floating point exception register and prints a diagnostic for the last floating point exception.

**SEE ALSO**

signal (II)

**BUGS**

Since the system does not save the exception register in a core image file, the message refers to the last error encountered by anyone. Floating exceptions are therefore volatile.

**NAME**

pr – print file

**SYNOPSIS**

**pr** [ **–h** *header* ] [ –*n* ] [ **+***n* ] [ **–w***n* ] [ –**l***n* ] [ **–t** ] [ **–s***c* ] [ **–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

Interconsole 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

**BUGS**

## NAME

prof – display profile data

## SYNOPSIS

**prof** [ **–v** ] [ **–a** ] [ **–l** ] [ file ]

## DESCRIPTION

*Prof* interprets the file *mon.out* produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file *(a.out* default) is read and correlated with the *mon.out* profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **–a** option is used, all symbols are reported rather than just external symbols. If the **–l** option is used, the output is listed by symbol value rather than decreasing percentage. If the **–v** option is used, all printing is suppressed and a profile plot is produced on the 611 display.

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 |
| /dev/vt0 | for plotting |

## SEE ALSO

monitor (III), profil (II), cc (I)

## BUGS

Beware of quantization errors.

**NAME**

ps − process status

**SYNOPSIS**

**ps** [ **aklx** ] [ namelist ]

**DESCRIPTION**

*Ps* prints certain indicia about active processes. The **a** flag asks for information about all processes with typewriters (ordinarily only one's own processes are displayed); **x** asks even about processes with no typewriter; **l** asks for a long listing. Ordinarily only the typewriter number (if not one's own), the process number, and an approximation to the command line are given. If the **k** flag is specified, the file */usr/sys/core* is used in place of */dev/mem.* This is used for post-mortem system debugging. If a second argument is given, it is taken to be the file containing the system's namelist.

The long listing is columnar and contains

The name of the process's control typewriter.

Flags associated with the process. 01: in core; 02: system process; 04: locked in code (e.g. for physical I/O); 10: being swapped; 20: being traced by another process.

The state of the process. 0: nonexistent; S: sleeping; W: waiting; R: running; Z: terminated; T: stopped.

The user ID of the process owner.

The process ID of the process; as in certain cults it is possible to kill a process if you know its true name.

The priority of the process; high numbers mean low priority.

The size in blocks of the core image of the process.

The event for which the process is waiting or sleeping; if blank, the process is running.

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 |
| /usr/sys/core | alternate core file |
| /dev | searched to find swap device and typewriter names |

**SEE ALSO**

kill (I)

**BUGS**

-

**NAME**

pwd – working directory name

**SYNOPSIS**

**pwd**

**DESCRIPTION**

*Pwd* prints the pathname of the working (current) directory.

**SEE ALSO**

chdir (I)

**BUGS**

**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** and **–c.**

**–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 Rational Fortran''.
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**

rev – reverse lines of a file

**SYNOPSIS**

**rev**

**DESCRIPTION**

*Rev* copies the standard input to the standard output, reversing the order of characters in every line.

**BUGS**

**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 a file has no write permission, *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 typewriter.

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**

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 runoff programs, but is utterly frozen.

-

## 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. |
| .n1 | 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**

sh – shell (command interpreter)

**SYNOPSIS**

**sh** [ **–t** ] [ **–c** ] [ name [ arg1 ... [ arg9 ] ] ]

**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 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.

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 non-directory file 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.* 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 '&'. The semicolon designates sequential execution. The ampersand causes the preceding pipeline to be executed without waiting for it to finish. The process id of such a pipeline is reported, so that it may be used if necessary for a subsequent *wait* or *kill.*

**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

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 before any redirection. Thus filters may write diagnostics to a location where they have a chance to be seen.

**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 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.

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’.

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 argument up to the last ‘/’ before a ‘*’ or ‘?’. 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.

**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.

Sequences of characters enclosed in double (") or single (´) quotes are also taken literally. 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*.

**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 [ name [ arg1 ... [ arg9 ] ] ]
```

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 (not 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*.

The argument '–t,' used alone, causes *sh* to read the standard input for a single line, execute it as a command, and then exit. This facility replaces the older 'mini-shell.' 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.

**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.

**Special commands.** The following commands are treated specially by the Shell.

*chdir* is done without spawning a new process by executing *sys chdir* (II).

*login* is done by executing /bin/login without creating a new process.

*wait* is done without spawning a new process by executing *sys wait* (II).

*shift* is done by manipulating the arguments to the Shell.

'**:**' is simply ignored.

**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.

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 the zero length file /dev/null.

**FILES**

/etc/glob, which interprets '*', '?', and '['.
/dev/null as a source of end-of-file.

**SEE ALSO**

'The UNIX Time-Sharing System', CACM, July, 1974, which gives the theory of operation of the Shell.
chdir (I), login (I), wait (I), shift (I)

**BUGS**

There is no way to redirect the diagnostic output.

**NAME**

shift – adjust Shell arguments

**SYNOPSIS**

**shift**

**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

```
: loop
if $1x = x exit
pr –3 $1
shift
goto loop
```

prints each of its arguments in 3-column format.

*Shift* is executed within the Shell.

**SEE ALSO**

sh (I)

**BUGS**

-

**NAME**

size – size of an object file

**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.

**BUGS**

**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:

```
: loop
command
sleep 37
goto loop
```

**SEE ALSO**

sleep (II)

**BUGS**

*Time* must be less than 65536 seconds.

**NAME**

sort, usort – sort or merge files

**SYNOPSIS**

**sort** [ **−abdnrt***x* ] [ **+***pos* [ *−pos* ] ] . . . [ **−mo** ] [ name ] . . .
**usort** [ **−umo** ] [ name ] . . .

**DESCRIPTION**

*Sort* sorts 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 in ASCII collating sequence, except that lower-case letters are considered the same as the corresponding upper-case letters. Non-ASCII bytes are ignored. The ordering is affected by the flags −**abdnrt**, one or more of which may appear:

**a** Do not map lower case letters.

**b** Leading blanks (spaces and tabs) are not included in fields.

**d** 'Dictionary' order: only letters, digits and blanks are significant in ASCII 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,* where *m* specifies a number of fields to skip, and *n* a number of characters to skip further into the next field. A missing 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). The ordering rule may be overridden for a particular key by appending one or more of the flags **abdnr** to **+***pos*.

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, a field is a string ending with a tab character.

When keys are specified, later keys are compared only when all earlier ones compare equal. Lines that 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**.

*Usort* is a somewhat specialized version of *sort* which accepts no collating sequence options: order is always plain ASCII. It also strips out the second and following copies of duplicated lines. A *u* flag prevents this stripping. *Usort* also understands the *m* and *o* options in the same way as *sort.*

**FILES**

/usr/tmp/stm???

**BUGS**

## NAME

spell − find spelling errors

## SYNOPSIS

**spell** [ −**v** ] file ...

## DESCRIPTION

*Spell* collects the words from the named documents, and looks them up in a dictionary. The words not found are printed on the standard output. Words which are reasonable transformations of dictionary entries (e.g. a dictionary entry plus *s* ) are not printed. If no files are given, the input is from the standard input.

If the −**v** flag is given, all words which are not literally in the dictionary are printed; those which can be transformed to lie in the dictionary are so marked, and the others are marked with asterisks.

The process takes several minutes.

## FILES

/usr/lib/w2006, /usr/dict/words, /usr/lib/spell[123]

## SEE ALSO

typo (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.

More suffixes, and perhaps some prefixes, should be added.

The dictionary cannot be distributed because of copyright limitations.

-

## 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.

## BUGS

-

## 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 the same as use of the –**s** option of *ld.*

## FILES

| | |
|---|---|
| /tmp/stm? | temporary file |

## SEE ALSO

ld (I), as (I)

## BUGS

-

## NAME

stty − set typewriter options

## SYNOPSIS

**stty** [ option ... ]

## DESCRIPTION

*Stty* sets certain I/O options on the current output typewriter. 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 below) |
| **nl0 nl1 nl2 nl3** | select style of delay for linefeed (see below) |
| **tab0 tab1 tab2 tab3** | select style of delay for tab (see below) |
| **ff0 ff1** | select style of delay for form feed (see below) |
| **tty33** | set all modes suitable for Teletype model 33 |
| **tty37** | set all modes suitable for Teletype model 37 |
| **vt05** | set all modes suitable for DEC VT05 terminal |
| **tn300** | set all modes suitable for GE Terminet 300 |
| **ti700** | set all modes suitable for Texas Instruments 700 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 typewriter baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface). |

The various delay algorithms are tuned to various kinds of terminals. In general the specifications ending in '0' mean no delay for the corresponding character.

## SEE ALSO

stty (II)

## BUGS

-

## NAME

tee – pipe fitting

## SYNOPSIS

**tee** [ name ... ]

## DESCRIPTION

*Tee* transcribes the standard input to the standard output and makes copies in the named files.

## BUGS

## 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 execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on the diagnostic output stream.

## 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 *sh* 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

troff – format text

## SYNOPSIS

**troff** [ **+***n* ] [ –*n* ] [ **−s***n* ] [ **−n***n* ] [ **−r***an* ] [ **−m***name* ] [ **–t** ] [ –**f** ] [ –**w** ] [ –**a** ] [ **−p***n* ] files

## DESCRIPTION

*Troff* formats text for a Graphic Systems phototypesetter according to control lines embedded in the text files. It reads the standard input if no file arguments are given. An argument of just ‘‘−’’ refers to the standard input. The non-file option arguments are interpreted as follows:

| | |
|---|---|
| **+***n* | Commence typesetting at the first page numbered *n* or larger. |
| –*n* | Stop after page *n.* |
| **−s***n* | Print output in groups of *n* pages, stopping the typesetter after each group. |
| **−n***n* | First generated (not necessarily printed) page is given the number *n;* simulates ‘‘.pn *n*’’. |
| **−r***an* | Set number register *a* to the value *n.* |
| **−m***name* | Prepends a standard macro file; simulates ‘‘.so /usr/lib/tmac.*name*’’. |
| –**t** | Place output on standard output instead of the phototypesetter. |
| –**f** | Refrain from feeding out paper and stopping the phototypesetter at the end. |
| –**w** | Wait until phototypesetter is available, if currently busy. |
| –**a** | Send a printable approximation of the results to the standard output. |
| **−p***n* | Print all characters with point-size *n* while retaining all prescribed spacings and motions. |

## FILES

| | |
|---|---|
| /usr/lib/suftab | suffix hyphenation tables |
| /tmp/rtm? | temporary |
| /usr/lib/tmac.* | standard macro files |

## SEE ALSO

TROFF User’s Manual (internal memorandum).
TROFF Made Trivial (internal memorandum).
nroff (I), eqn (I), catsim (VI)

## BUGS

-

**NAME**

tty – get typewriter name

**SYNOPSIS**

**tty**

**DESCRIPTION**

*Tty* gives the name of the user's typewriter 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 typewriter.

**BUGS**

**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

**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.

**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)

**BUGS**

**NAME**

wait – await completion of process

**SYNOPSIS**

**wait**

**DESCRIPTION**

Wait until all processes started with **&** have completed, and report on abnormal terminations.

Because *sys wait* 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 from another terminal or to hang up.

## NAME

wc – word count

## SYNOPSIS

**wc** [ 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.

## BUGS

## NAME

who – who is on the system

## SYNOPSIS

**who** [ who-file ] [ **am I** ]

## DESCRIPTION

*Who*, without an argument, lists the name, typewriter 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, typewriter 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 typewriter. 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)

## BUGS

**NAME**

write – write to another user

**SYNOPSIS**

**write** user [ ttyno ]

**DESCRIPTION**

*Write* copies lines from your typewriter 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 typewriter 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 typewriter name.

Permission to write may be denied or granted by use of the *mesg* command. At the outset writing is allowed. Certain commands, in particular *roff* 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 him to write back before starting to send. Each party should end each message with a distinctive signal ( **(o)** for ‘over’ is conventional) that the other may reply. **(oo)** (for ‘over and out’) is suggested when conversation is about to be terminated.

**FILES**

| | |
|---|---|
| /etc/utmp | to find user |
| /bin/sh | to execute ‘!’ |

**SEE ALSO**

mesg (I), who (I), mail (I)

**BUGS**

-

**NAME**

yacc – yet another compiler-compiler

**SYNOPSIS**

**yacc** [ –**vor** ] [ 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 –**o** flag calls an optimizer for the tables; the optimized tables, with parser included, appear on file *y.tab.c*

The –**r** flag causes Yacc to accept grammars with Ratfor actions, and produce Ratfor output on *y.tab.r;* –**r** implies the –**o** flag. Typical usage is then

```
rc y.tab.r other.o
```

**SEE ALSO**

‘‘LR Parsing’’, by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974. ‘‘The YACC Compiler-compiler’’, internal memorandum.

**AUTHOR**

S. C. Johnson

**FILES**

| | |
|---|---|
| y.output | |
| y.tab.c | |
| y.tab.r | when ratfor output is obtained |
| yacc.tmp | when optimizer is called |
| /lib/liby.a | runtime library for compiler |
| /usr/yacc/fpar.r | ratfor parser |
| /usr/yacc/opar.c | parser for optimized tables |
| /usr/yacc/yopti | optimizer postpass |

**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 dialled in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 512 bytes (counting the null at the end of each argument) is presented to *exec.*

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with one of the magic numbers 407 or 410.

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 – (unused)

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 typewriter
The file mentioned in *stty* or *gtty* is not a typewriter 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**

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

sort − sort or merge files

## SYNOPSIS

**sort** [ −**abdnrt***x* ] [ +*pos* [ −*pos* ] ] . . . [ −**mo** ] [ name ] . . .

## DESCRIPTION

*Sort* sorts 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 in ASCII collating sequence, except that lower-case letters are considered the same as the corresponding upper-case letters. Non-ASCII bytes are ignored. The ordering is affected by the flags −**abdnrt**, one or more of which may appear:

**a** Do not map lower case letters.

**b** Leading blanks (spaces and tabs) are not included in fields.

**d** 'Dictionary' order: only letters, digits and blanks are significant in ASCII 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,* where *m* specifies a number of fields to skip, and *n* a number of characters to skip further into the next field. A missing 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). The ordering rule may be overridden for a particular key by appending one or more of the flags **abdnr** to +*pos*.

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, a field is a string ending with a tab character.

When keys are specified, later keys are compared only when all earlier ones compare equal. Lines that 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**.

## FILES

/usr/tmp/stm???

-

**NAME**

chown – change owner and group of a file

**SYNOPSIS**

(chmod = 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. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

**SEE ALSO**

chown (VIII), chgrp (VIII), 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.; not in assembler)
**sys csw**

**getcsw( )**

**DESCRIPTION**

The setting of the console switches is returned (in r0).

**NAME**

dup – duplicate an open file descriptor

**SYNOPSIS**

(dup = 41.; not in assembler)
(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)

**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 512 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 512 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 the process's files 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)

## DIAGNOSTICS

None.

## 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)

## 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 (not just negative) 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.; not in assembler)
**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)

**DIAGNOSTICS**

–

**NAME**

getpid – get process identification

**SYNOPSIS**

(getpid = 20.; not in assembler)
**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.

**SEE ALSO**

–

**DIAGNOSTICS**

–

**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)

**DIAGNOSTICS**

–

-

**NAME**

gtty – get typewriter 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 typewriter 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 typewriter. From C, a −1 value is returned for an error, 0, for a successful call.

## NAME

indir − indirect system call

## SYNOPSIS

(indir = 0.; not in assembler)
**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.

## SEE ALSO

−

## DIAGNOSTICS

−

**NAME**

kill – send signal to a process

**SYNOPSIS**

(kill = 37.; not in assembler)
(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, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes which have the same controlling typewriter and user ID.

In no case is it possible for a process to kill itself.

**SEE ALSO**

signal (II), kill (I)

**DIAGNOSTICS**

The error bit (c-bit) is set if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist. From C, −1 is returned.

**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**

link (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**

mknod – make a directory or a special file

**SYNOPSIS**

(mknod = 14.; not in assembler)
**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.

**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.

**BUGS**

–

**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 −220. 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 119 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**

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.

**BUGS**

**NAME**

profil – execution time profile

**SYNOPSIS**

(profil = 44.; not in assembler)
**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)

**DIAGNOSTICS**

–

-

**NAME**

ptrace – process trace

**SYNOPSIS**

(ptrace = 26.; not in assembler)
(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 data space (normally used); 2 indicates the instruction space (when I and D space are separated). *addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system’s per-process data area corresponding to *addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5 The given *data* is written at the word in the process’s address space corresponding to *addr,* which must be even. No useful value is returned. Request 4 specifies data space (normally used), 5 specifies instruction 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 typewriter 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)

**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.; not in assembler)
(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**

signal – catch or ignore signals

**SYNOPSIS**

(signal = 48.)
**sys signal; sig; label**
(old value in r0)

**signal(sig, func)**
**int (*func)( );**

**DESCRIPTION**

A *signal* is generated by some abnormal event, initiated either by user at a typewriter (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but this call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals:

| | |
|---|---|
| 1 | hangup |
| 2 | interrupt |
| 3* | quit |
| 4* | illegal instruction (not reset when caught) |
| 5* | trace trap (not reset when caught) |
| 6* | IOT instruction |
| 7* | EMT instruction |
| 8* | floating point exception |
| 9 | kill (cannot be caught or ignored) |
| 10* | bus error |
| 11* | segmentation violation |
| 12* | bad argument to system call |
| 13 | write on a pipe with no one to read it |

In the assembler call, if *label* is 0, the process is terminated when the signal occurs; this is the default action. If *label* is odd, the signal is ignored. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt. Except as indicated, a signal is reset to 0 after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

In C, if *func* is 0, the default action for signal *sig* (termination) is reinstated. If *func* is 1, the signal is ignored. If *func* is non-zero and even, it is assumed to be the address of a function entry point. When the signal occurs, the function will be called. A return from the function will continue the process at the point it was interrupted. As in the assembler call, *signal* must in general be called again to catch subsequent signals.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a *read* or *write* on a slow device (like a typewriter; but not a file); and during or *wait.* When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned a characteristic error status. The user's program may then, if it wishes, re-execute the call.

The starred signals in the list above cause a core image if not caught or ignored.

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; 0 indicates success.

-

**BUGS**

**NAME**

sleep – stop execution for interval

**SYNOPSIS**

(sleep = 35.; not in assembler)
(seconds in r0)
**sys sleep**

**sleep(seconds)**

**DESCRIPTION**

The current process is suspended from execution for the number of seconds specified by the argument.

**SEE ALSO**

sleep (I)

**DIAGNOSTICS**

–

**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 typewriter

## 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 typewriter whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the typewriter 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. If DC-11, DL-11 or KL-11 interfaces are used, impossible speed changes are ignored.

| | |
|---|---|
| 0 | (hang up dataphone) |
| 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, in that the code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program, and the half-duplex line discipline required for the 202 dataset (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 which determine the system's treatment of the typewriter:

| | |
|---|---|
| 100000 | Select one of two algorithms for backspace delays |
| 040000 | Select one of two algorithms for form-feed and vertical-tab delays |
| 030000 | Select one of four algorithms for carriage-return delays |
| 006000 | Select one of four algorithms for tab delays |
| 001400 | Select one of four algorithms for new-line delays |
| 000200 | even parity allowed on input (e. g. for M37s) |
| 000100 | odd parity allowed on input |
| 000040 | raw mode: wake up on all characters |
| 000020 | map CR into LF; echo LF or CR as CR-LF |

| | |
|---|---|
| 000010 | echo (full duplex) |
| 000004 | map upper case to lower on input (e. g. M33) |
| 000002 | echo and print tabs as spaces |
| 000001 | hang up (remove ‘data terminal ready,’ lead CD) 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 ignored but will be used for Terminet 300’s.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is unimplemented and is 0.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37’s. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Other types are unimplemented and are 0.

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file 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 GE TermiNet 300’s and other terminals without the newline function).

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 typewriters, but since none of them are part of the standard system the specifications will not be given.

## SEE ALSO

stty (I), gtty (II)

## DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a negative value indicates an error.

**NAME**

sync – update super-block

**SYNOPSIS**

(sync = 36.; not in assembler)
**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 (VIII), update (VIII)

**DIAGNOSTICS**

–

-

**NAME**

time – get date and time

**SYNOPSIS**

(time = 13.)
**sys time**

**time(tvec)**
**int tvec[2];**

**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)

**DIAGNOSTICS**

–

-

**NAME**

times – get process times

**SYNOPSIS**

(times = 43.; not in assembler)
**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 {
        int     proc_user_time;
        int     proc_system_time;
        int     child_user_time[2];
        int     child_system_time[2];
};
```

The children times are the sum of the children's process times and their children's times.

**SEE ALSO**

time (I)

**DIAGNOSTICS**

–

**BUGS**

The process times should be 32 bits as well.

**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).

**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

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**

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.

**BUGS**

**NAME**

abs, fabs – absolute value

**SYNOPSIS**

**abs(i)**
**int i;**

**double fabs(x)**
**double x;**

**DESCRIPTION**

*Abs* returns the absolute value of its integer operand; *fabs* is the *double* version.

-

## 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 –π/2 to π/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 –π to π.

**DIAGNOSTIC**

There is no error return.

**BUGS**

## 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 signalled.

-

## 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

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 eight bytes of ASCII alphanumerics in a global cell called ‘‘word’’.

From C, the *key* argument is a string and the value returned is a pointer to the eight-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 iff 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)

## BUGS

## 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)

## BUGS

**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+...);
```

(see *break* (II)). The problem with this is that it ignores any other subroutines which may want to extend core for their purposes; these include *sbrk* (see *break* (II)), *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)

**BUGS**

## 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.

## BUGS

-

**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**.

**BUGS**

-

**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 $0 \leqq f < y$.

**BUGS**

**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), cc (I) ('−f' option)

**BUGS**

Rounding mode is not interpreted. It's slow.

**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)

**BUGS**

-

**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.

**SEE ALSO**

open (II), read (II), getchar (III), putc (III)

**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 introduction (II)).

**BUGS**

**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.

**SEE ALSO**

getc (III)

**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.

**BUGS**

## 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.)

## BUGS

**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.

## BUGS

-

## NAME

monitor – prepare execution profile

## SYNOPSIS

**monitor(lowpc, highpc, buffer, bufsize)**
**int lowpc( ), highpc( ), buffer[ ], bufsize;**

## DESCRIPTION

*Monitor* is an interface to the system's profile entry (II). *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;
} nl[ ];
```

**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.

**BUGS**

**NAME**

perror, sys_errlist, sys_nerr, errno – system error 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**

Introduction to System Calls

**BUGS**

**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.

**BUGS**

-

**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.

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**.**ddd**e**±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.

l 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).

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.

## SEE ALSO

creat (II), write (II), getc (III)

## 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 introduction (II).

## BUGS

-

**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.

**SEE ALSO**

putc (III)

**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)

**BUGS**

**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**

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)

**BUGS**

**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**

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.

**BUGS**

**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**

ttyn – return name of current typewriter

**SYNOPSIS**

**jsr　　pc,ttyn**

**ttyn(file)**

**DESCRIPTION**

*Ttyn* hunts up the last character of the name of the typewriter which is the standard input (from *as*) or is specified by the argument *file* descriptor (from C). If *n* is returned, the typewriter name is then ‘‘/dev/tty*n*’’.

**x** is returned if the indicated file does not correspond to a typewriter.

**BUGS**

-

## 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), Graphic Systems specification (available on request)

## BUGS

-

**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[f-u]

**SEE ALSO**

tty (IV), stty (II)

**BUGS**

## NAME

dn – DN-11 ACU interface

## DESCRIPTION

The *dn?* files are write-only. The permissible codes are:

| | |
|---|---|
| 0-9 | dial 0-9 |
| : | dial * |
| ; | dial # |
| – | 4 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/dn0 | connected to 801 with dp0 |
| /dev/dn1 | not currently connected |
| /dev/dn2 | not currently connected |

## SEE ALSO

dp (IV)

## BUGS

**NAME**

dp – DP-11 201 data-phone interface

**DESCRIPTION**

The *dp0* file is a 201 data-phone interface. *Read* and *write* calls to dp0 are limited to a maximum of 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 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.

**FILES**

/dev/dp0

**SEE ALSO**

dn (IV), gerts (III)

**BUGS**

**NAME**

hp – RH-11/RP04 moving-head disk

**DESCRIPTION**

The files *hp0* ... *hp7* refer to sections of RP disk drive 0. The files *hp8* ... *hp15* refer to drive 1 etc. This is done since the size of a full RP drive is 170,544 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 pseudo-disks on each drive are as follows:

| disk | start | length |
|---|---|---|
| 0 | 0 | 9614 |
| 1 | 18392 | 65535 |
| 2 | 48018 | 65535 |
| 3 | 149644 | 20900 |
| 4 | 0 | 40600 |
| 5 | 41800 | 40600 |
| 6 | 83600 | 40600 |
| 7 | 125400 | 40600 |

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 *hp* 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 *rhp* and end with a number which selects the same disk section as the corresponding *hp* 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/hp?, /dev/rhp?

**BUGS**

-

**NAME**

hs − RH11/RS03-RS04 fixed-head disk file

**DESCRIPTION**

The files *hs0* ... *hs7* refer to RJS03 disk drives 0 through 7. The files *hs8* ... *hs15* refer to RJS04 disk drives 0 through 7. The RJS03 drives are each 1024 blocks long and the RJS04 drives are 2048 blocks long.

The *hs* 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'' inteface 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 *rhs.* 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/hs?, /dev/rhs?

**BUGS**

**NAME**

ht – RH-11/TU-16 magtape interface

**DESCRIPTION**

The files *mt0, ..., mt7* refer to the DEC RH/TM/TU16 magtape. When opened for reading or writing, the tape is rewound. When closed, it is rewound; if it was open for writing, a double end-of-file is written first.

A standard tape consists of a series of 512 byte records terminated by a double end-of-file. 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. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

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, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

**FILES**

/dev/mt?, /dev/rmt?

**SEE ALSO**

tp (I)

**BUGS**

Raw I/O doesnt work yet. The magtape system is supposed to be able to take 64 drives. Such addressing has never been tried. These bugs will be fixed when we get more experience with this device.

If any non-data error is encountered, it refuses to do anything more until closed. In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

**NAME**

kl – KL-11 or DL-11 asynchronous interface

**DESCRIPTION**

The discussion of typewriter 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 typewriter 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 typewriter.

**FILES**

/dev/tty8console

**SEE ALSO**

tty (IV), init (VIII)

**BUGS**

Modem control for the DL-11E is not implemented.

-

## NAME

lp – line printer

## DESCRIPTION

*Lp* provides the interface to any of the standard DEC 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:

| | |
|---|---|
| { | (̶ |
| } | )̶ |
| \` | $\underline{\acute{}}$ |
| \| | !̶ |
| ~ | $\underline{\hat{}}$ |

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 printout if it is unpleasantly near the left margin.

## FILES

/dev/lp

## SEE ALSO

lpr (I)

## BUGS

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 an 18-bit quantity which is used directly as a UNIBUS address. 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.

The file *kmem* is the same as *mem* except that kernel virtual memory 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**

pc – PC-11 paper tape reader/punch

**DESCRIPTION**

*Ppt* refers to the PC-11 paper tape reader or punch, depending on whether it is read or written.

When *ppt* is opened for writing, a 100-character leader is punched. Thereafter each byte written is punched on the tape. No editing of the characters is performed. When the file is closed, a 100-character trailer is punched.

When *ppt* is opened for reading, the process waits until tape is placed in the reader and the reader is on-line. Then requests to read cause the characters read to be passed back to the program, again without any editing. This means that several null leader characters will usually appear at the beginning of the file. Likewise several nulls are likely to appear at the end. End-of-file is generated when the tape runs out.

Seek calls for this file are meaningless.

**FILES**

/dev/ppt

**BUGS**

If both the reader and the punch are open simultaneously, the trailer is sometimes not punched. Sometimes the reader goes into a dead state in which it cannot be opened.

-

**NAME**

rf – RF11/RS11 fixed-head disk file

**DESCRIPTION**

This file refers to the concatenation of all RS-11 disks.

Each disk contains 1024 256-word blocks. The length of the combined RF file is 1024×(minor+1) blocks. That is minor device zero is taken to be 1024 blocks long; minor device one is 2048, etc.

The *rf0* file accesses 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 name of the raw RF file is *rrf0.* 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/rf0, /dev/rrf0

**BUGS**

The 512-byte restrictions on the raw device are not physically necessary, but are still imposed.

**NAME**

rk – RK-11/RK03 (or RK05) disk

**DESCRIPTION**

*Rk?* refers to an entire RK03 disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871.

Drive numbers (minor devices) of eight and larger are treated specially. Drive 8+*x* is the *x*+1 way interleaving of devices rk0 to rk*x*. Thus blocks on rk10 are distributed alternately among rk0, rk1, and rk2.

The *rk* files discussed above 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 RK files begin with *rrk* and end with a number which selects the same disk as the corresponding *rk* 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/rk?, /dev/rrk?

**BUGS**

Care should be taken in using the interleaved files. First, the same drive should not be accessed simultaneously using the ordinary name and as part of an interleaved file, because the same physical blocks have in effect two different names; this fools the system's buffering strategy. Second, the combined files cannot be used for swapping or raw I/O.

**NAME**

rp – RP-11/RP03 moving-head disk

**DESCRIPTION**

The files *rp0 ... rp7* refer to sections of RP disk drive 0. The files *rp8 ... rp15* refer to drive 1 etc. This is done since the size of a full RP drive 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 pseudo-disks on each drive are as follows:

| disk | start | length |
|---|---|---|
| 0 | 0 | 40600 |
| 1 | 40600 | 40600 |
| 2 | 0 | 9200 |
| 3 | 72000 | 9200 |
| 4 | 0 | 65535 |
| 5 | 15600 | 65535 |
| 6-7 | unassigned | |

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. Here is a suggestion for two useful configurations: If the root of the file system is on some other device and the RP used as a mounted device, then *rp0* and *rp1,* which divide the disk into two equal size portions, is a good idea. Other things being equal, it is advantageous to have two equal-sized portions since one can always be copied onto the other, which is occasionally useful.

If the RP is the only disk and has to contain the root and the swap area, the root can be put on *rp2* and a mountable file system on *rp5.* Then the swap space can be put in the unused blocks 9200 to 15600 of *rp0* (or, equivalently, *rp4).* This arrangement puts the root file system, the swap area, and the i-list of the mounted file system relatively near each other and thus tends to minimize head movement.

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?

**BUGS**

-

**NAME**

tc – TC-11/TU56 DECtape

**DESCRIPTION**

The files *tap0* ... *tap7* refer to the TC-11/TU56 DECtape drives 0 to 7.

The 256-word blocks on a standard DECtape are numbered 0 to 577.

**FILES**

/dev/tap?

**SEE ALSO**

tp (I)

**BUGS**

**NAME**

tm – TM-11/TU-10 magtape interface

**DESCRIPTION**

The files *mt0, ..., mt7* refer to the DEC TU10/TM11 magtape. When opened for reading or writing, the tape is rewound. When closed, it is rewound; if it was open for writing, an end-of-file is written first.

A standard tape consists of a series of 512 byte records terminated by an end-of-file. 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. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

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, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

**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. In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

**NAME**

tty – general typewriter interface

**DESCRIPTION**

This section describes both a particular special file, and the general nature of the typewriter interface.

The file */dev/tty* is, in each process, a synonym for the control typewriter associated with that process. It is useful for programs or Shell sequences which wish to be sure of writing messages on the typewriter 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 typewriter is currently in use.

As for typewriters 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; the KL, DC, and DH writeups (IV) describe peculiarities of the individual devices.

When a typewriter file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by *init* and become a user's input and output file. The very first typewriter file open in a process becomes the *control typewriter* for that process. The control typewriter plays a special role in handling quit or interrupt signals, as discussed below. The control typewriter 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* system call (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; 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, typewriter 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 |
|---|---|
| ` | \´ |
| \| | \! |

| | |
|---|---|
| ~ | \ˆ |
| { | \( |
| } | \) |

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, but parity is still generated for output characters.

The ASCII EOT (control-D) character may be used to generate an end of file from a typewriter. 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 dataset drops (usually because the user has hung up his terminal) a *hangup* signal is sent to all processes with the typewriter as control typewriter. 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 typewriter 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 with the associated control typewriter. 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 character FS 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. If you find it hard to type this character, try control-\ or control-shift-L.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is always generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals which respond to it from hanging up.

## FILES

/dev/tty

## SEE ALSO

dc (IV), kl (IV), dh (IV), getty (VIII), stty (I, II), gtty (I, II), signal (II)

## BUGS

Half-duplex terminals are not supported. On raw-mode output, parity should be transmitted as specified in the characters written.

**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 177555(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 16 bytes long:

| | |
|---|---|
| 0-7 | file name, null padded on the right |
| 8-11 | modification time of the file |
| 12 | user ID of file owner |
| 13 | file mode |
| 14-15 | file size |

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size give reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

**SEE ALSO**

ar (I), ld (I)

**BUGS**

Names are only 8 characters, not 14. More important, there isn't enough room to store the proper mode, so *ar* always extracts in mode 666.

-

**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 ˆ | 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**

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 1024 bytes of the core image are a copy of the system’s per-user data for the process, including the registers as they were at the time of the fault. The remainder represents the actual contents of the user’s core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

The format of the information in the first 1024 bytes 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) |
| r0 | 1772 | |
| r1 | 1766 | |
| r2 | 1750 | |
| r3 | 1752 | |
| r4 | 1754 | |
| r5 | 1756 | |
| sp | 1764 | |
| pc | 1774 | |
| ps | 1776 | |

In general the debuggers *db (I)* and *cdb (I)* are sufficient to deal with core images.

**SEE ALSO**

cdb (I), db (I), signal (II)

-

**NAME**

dir – 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.

By convention, the first two entries in each directory are for ‘‘**.**’’ and ‘‘**..**’’. The first is an entry for the directory itself. The second is for the parent directory. The meaning of ‘‘**..**’’ is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases ‘‘**..**’’ has the same meaning as ‘‘**.**’’.

**SEE ALSO**

file system (V)

**NAME**

dump – incremental dump tape format

**DESCRIPTION**

The *dump* and *restor* 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 file system (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 file system (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), file system(V)

**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 common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the *super block.* Starting from its first word, the format of a super-block is

```
struct {
        int     isize;
        int     fsize;
        int     nfree;
        int     free[100];
        int     ninode;
        int     inode[100];
        char    flock;
        char    ilock;
        char    fmod;
        int     time[2];
};
```

*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.*

*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.

*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.

*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.

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$^{\bullet}$((*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 word 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 7, 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.

If *i* is equal to 7, then the file has become extra-large (huge), and *addr[7]* is the address of a first indirect block. Each word in this block is the number of a second-level indirect block; each word in the second-level indirect blocks points to a data block. Notice that extra-large files are not marked by any mode bit, but only by having *addr[7]* non-zero; and that although this scheme allows for more than 256×256×512 = 33,554,432 bytes per file, the length of files is stored in 24 bits so in practice a file can be at most 16,777,216 bytes long.

For block *b* in a file to exist, it is not necessary that all blocks less than *b* exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

-

**SEE ALSO**

icheck, dcheck (VIII)

-

## 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 model 37 Teletypes 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 | ¬ | _ |
| partial | ∂ | ] | integral | ∫ | ˆ | | | |

## SEE ALSO

ascii (VII)

**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**

mtab – mounted file system table

**DESCRIPTION**

*Mtab* resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last ‘‘/’’ is thrown away.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

**FILES**

/etc/mtab

**SEE ALSO**

mount (VIII), umount (VIII)

**BUGS**

**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
numerical group ID (for now, always 1)
GCOS job number, box number, optional GCOS user-id
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 GCOS field is used only when communicating with that system, and in other installations can contain any desired information. 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**

tabs – set tab stops

**SYNOPSIS**

**cat /usr/pub/tabs**

**DESCRIPTION**

Printing this file on a suitable terminal sets tab stops every 8 columns. Suitable terminals include the Teletype model 37 and the GE TermiNet 300.

These tab stop settings are desirable because UNIX assumes them in calculating delays.

**NAME**

tp – DEC/mag tape formats

**DESCRIPTION**

The command *tp* dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See boot procedures (VIII).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 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 (file system (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 25 (resp. 63) on are available for file storage.

A fake entry (see tp (I)) has a size of zero.

**SEE ALSO**

file system (V), tp (I)

## NAME

ttys – typewriter initialization data

## DESCRIPTION

The *ttys* file is read by the *init* program and specifies which typewriter 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 typewriter; 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 to 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 typewriter 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 typewriter. Furthermore, the typewriter name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with typewriter 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**

**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 programs'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 five lines. The first line gives a year, month number, 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 two numbers which specify options to the program governing the completeness of the report; they are ordinarily both ‘‘1’’. The first option flag suppresses output when the sky is not dark; the second supresses 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 three are the elements themselves (including certain derivatives of the elements).

**FILES**

/usr/jfo/el/* – orbital element files

**SEE ALSO**

sky (VI)

**AUTHOR**

J. F. Ossanna

-

**BUGS**

**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.

**BUGS**

**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**

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**

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.

**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.

**BUGS**

-

## NAME

factor – discover prime factors of a number

## SYNOPSIS

**factor**

## DESCRIPTION

When *factor* is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than $2^{56}$ (about $7.2\times10^{16}$) 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.

If *factor* is invoked with an argument, it factors the number as above and then exits.

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.

## BUGS

**NAME**

fed – edit form letter memory

**SYNOPSIS**

**fed**

**DESCRIPTION**

*Fed* is used to edit a form letter associative memory file, **form.m,** which consists of named strings. Commands consist of single letters followed by a list of string names separated by a single space and ending with a new line. The conventions of the Shell with respect to '*' and '?' hold for all commands but **m**. The commands are:

**e** name ...
*Fed* writes the string whose name is *name* onto a temporary file and executes *ed.* On exit from the *ed* the temporary file is copied back into the associative memory. Each argument is operated on separately. Be sure to give an editor *w* command (without a filename) to rewrite *fed's* temporary file before quitting out of *ed.*

**d** [ name ... ]
deletes a string and its name from the memory. When called with no arguments **d** operates in a verbose mode typing each string name and deleting only if a **y** is typed. A **q** response returns to *fed*'s command level. Any other response does nothing.

**m** name1 name2 ...
(move) changes the name of name1 to name2 and removes previous string name2 if one exists. Several pairs of arguments may be given. Literal strings are expected for the names.

**n** [ name ... ]
(names) lists the string names in the memory. If called with the optional arguments, it just lists those requested.

**p** name ...
prints the contents of the strings with names given by the arguments.

**q**
returns to the system.

**c** [ **p** ] [ **f** ]
checks the associative memory file for consistency and reports the number of free headers and blocks. The optional arguments do the following:

**p** causes any unaccounted-for string to be printed.

**f** fixes broken memories by adding unaccounted-for headers to free storage and removing references to released headers from associative memory.

**FILES**

| | |
|---|---|
| /tmp/ftmp? | temporary |
| form.m | associative memory |

**SEE ALSO**

form (VI), ed (I), sh (I)

**WARNING**

It is legal but unwise to have string names with blanks, '*' or '?' in them.

**BUGS**

**NAME**

form − form letter generator

**SYNOPSIS**

**form** proto arg ...

**DESCRIPTION**

*Form* generates a form letter from a prototype letter, an associative memory, arguments and in a special case, the current date.

If *form* is invoked with the *proto* argument *x*, the associative memory is searched for an entry with name *x* and the contents filed under that name are used as the prototype. If the search fails, the message '[*x*]:' is typed on the console and whatever text is typed in from the console, terminated by two new lines, is used as the prototype. If the prototype argument is missing, '{letter}' is assumed.

Basically, *form* is a copy process from the prototype to the output file. If an element of the form [*n*] (where *n* is a digit from 1 to 9) is encountered, the *n*-th *arg* is inserted in its place, and that argument is then rescanned. If [0] is encountered, the current date is inserted. If the desired argument has not been given, a message of the form '[*n*]:' is typed. The response typed in then is used for that argument.

If an element of the form [*name*] or {*name*} is encountered, the *name* is looked up in the associative memory. If it is found, the contents of the memory under this *name* replaces the original element (again rescanned). If the *name* is not found, a message of the form '[*name*]:' is typed. The response typed in is used for that element. The response is entered in the memory under the name if the name is enclosed in [ ]. The response is not entered in the memory but is remembered for the duration of the letter if the name is enclosed in {}. Brackets and braces may be nested.

In both of the above cases, the response is typed in by entering arbitrary text terminated by two new lines. Only the first of the two new lines is passed with the text.

If one of the special characters [{]}\ is preceded by a \, it loses its special character.

If a file named 'forma' already exists in the user's directory, 'formb' is used as the output file and so forth to 'formz'.

The file 'form.m' is created if none exists. Because form.m is operated on by the disc allocator, it should only be changed by using *fed*, the form letter editor, or *form*.

**FILES**

| | |
|---|---|
| form.m | associative memory |
| form? | output file (read only) |

**SEE ALSO**

fed (VI), roff (I)

**BUGS**

An unbalanced ] or } acts as an end of file but may add a few strange entries to the associative memory.

**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* (VI). These plotters exist: *gsip,* for the GSI and other Diablo terminals; *tek,* for the Tektronix 4014 terminal; and *vt0* for the on-line storage scope.

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** Place character string given by next argument at 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).

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

**x** Next 1 (or 2) arguments are lower (and upper) *x* limits.

**y** Next 1 (or 2) arguments are lower (and upper) *y* limits.

**h** Next argument is fraction of space for height

**w** Next argument is fraction of space for width.

**r** Next argument is fraction of space to move right before plotting.

**u** Next argument is fraction of space to move up before plotting.

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. Grid lines and automatically determined limits fall on round values, however roundness may be subverted by giving an inappropriately rounded lower limit. Plotting symbols specified by **c** are placed so that a small initial letter, such as + o x, will fall approximately on the plotting point.

**SEE ALSO**

spline (VI), plot (VI)

**BUGS**

A limit of 1000 points is enforced silently.

-

**NAME**

gsi – interpret extended character set on GSI terminal

**SYNOPSIS**

**gsi**

**DESCRIPTION**

*Gsi* interprets special characters understood by the Model 37 Teletype terminal and turns them into the escape sequences understood by the GSI and other Diablo-based terminals. The things interpreted include vertical motions and extended graphic characters. It is most often used in a pipeline like

```
neqn file ... | nroff | gsi
```

**SEE ALSO**

greek (V)

**BUGS**

Some funny characters can't be correctly printed in column 1 because you can't move to the left from there.

**NAME**

m6 – general purpose macroprocessor

**SYNOPSIS**

**m6** [ name ]

**DESCRIPTION**

*M6* copies the standard input to the standard output, with substitutions for any macro calls that appear. When a file name argument is given, that file is read before the standard input.

The processor is as described in the reference with these exceptions:

*#def,arg1,arg2,arg3:* causes *arg1* to become a macro with defining text *arg2* and (optional) built-in serial number *arg3*.

*#del,arg1:* deletes the definition of macro *arg1*.

*#end:* is not implemented.

*#list,arg1:* sends the name of the macro designated by *arg1* to the current destination without recognition of any warning characters; *arg1* is 1 for the most recently defined macro, 2 for the next most recent, and so on. The name is taken to be empty when *arg1* doesn't make sense.

*#warn,arg1,arg2:* replaces the old warning character *arg1* by the new warning character *arg2*.

*#quote,arg1:* sends the definition text of macro *arg1* to the current destination without recognition of any warning characters.

*#serial,arg1:* delivers the built-in serial number associated with macro *arg1*.

*#source,arg1:* is not implemented.

*#trace,arg1:* with *arg1* = '1' causes a reconstruction of each later call to be placed on the standard output with a call level number; other values of *arg1* turn tracing off.

The built-in 'warn' may be used to replace inconvenient warning characters. The example below replaces '#' ':' '<' '>' by '[' ']' '{' '}'.

```
#warn,<#>,[:
[warn,<:>,]:
[warn,[substr,<<>>,1,1;,{]
[warn,[substr,{{>>,2,1;,}]
[now,{calls look like this}]
```

Every built-in function has a serial number, which specifies the action to be performed before the defining text is expanded. The serial numbers are: 1 gt, 2 eq, 3 ge, 4 lt, 5 ne, 6 le, 7 seq, 8 sne, 9 add, 10 sub, 11 mpy, 12 div, 13 exp, 20 if, 21 def, 22 copy, 23 warn, 24 size, 25 substr, 26 go, 27 gobk, 28 del, 29 dnl, 32 quote, 33 serial, 34 list, 35 trace. Serial number 0 specifies no built-in action.

**SEE ALSO**

A. D. Hall, M6 Reference Manual. Computer Science Technical Report #2, Bell Laboratories, 1969.

**DIAGNOSTICS**

Various table overflows and ''impossible'' conditions result in comment and dump. There are no diagnostics for poorly formed input.

**AUTHOR**

M. D. McIlroy

**BUGS**

Provision should be made to extend tables as needed, instead of wasting a big fixed core allocation. You get what the PDP11 gives you for arithmetic.

**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).

**BUGS**

-

**NAME**

plot: tek, gsip, vt0 – graphics filters

**SYNOPSIS**

source | **tek**
source | **gsip**
source | **vt0**

**DESCRIPTION**

These commands produce graphical output on the Tektronix 4014 terminal, the GSI or other Diablo-mechanism terminals, and the on-line storage scope respectively. They read the standard input to obtain plotting instructions, which are usually generated by a program calling the graphics subroutines described in *plot* (VII). Each instruction consists of an ASCII letter usually followed by binary information. A plotting coordinate is transmitted as four bytes representing the *x* and *y* values; each value is a signed number transmitted low-order byte first. The assumed plotting space is set by request. The instructions are taken from

m move: the next four bytes specify the coordinates of a point to move to. This is used before writing a label.

p point: the next four bytes specify the coordinates at which a point is drawn.

l line: the next eight bytes are taken as two pairs of coordinates specifying the endpoints of a line to be drawn.

t label: the bytes up to a new-line are written as ASCII starting at the last point drawn or moved to.

a arc: the first four bytes specify the center, the next four specify the starting point, and the last four specify 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. This command is not necessarily implemented on all (or even any) of the output devices.

c circle: The first four bytes specify the center of the circle, the next two the radius.

e erases the screen

f linemod: takes the following string as the type for all future lines. The types are ‘dotted,’ ‘solid,’ ‘longdashed,’ ‘shortdashed,’ and ‘dotdashed.’ This instruction is effective only with the Tektronix terminal.

d dotline: takes the first four bytes as the coordinates of the beginning of a dotted line. The next two are a signed x-increment, and the next two are a word count. Following are the indicated number of byte-pairs representing words. For each bit in this list of words a point is plotted which is visible if the bit is ‘1,’ invisible if not. Each point is offset rightward by the x-increment. The instruction is effective only on the vt0 scope.

**SEE ALSO**

plot (VII), graph (VI)

**BUGS**

-

**NAME**

primes – print all primes larger than somewhat

**SYNOPSIS**

**primes**

**DESCRIPTION**

When *primes* is invoked, it waits for a number to be typed in. If you type in a positive number less than $2^{56}$ (about $7.2 \times 10^{16}$) it will print all primes greater than or equal to this number.

**DIAGNOSTICS**

'Ouch.' for input out of range or for garbage input.

**BUGS**

-

## 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/*

## BUGS

## NAME

sky − obtain ephemerides

## SYNOPSIS

**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.*

## AUTHOR

R. Morris

## BUGS

## 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 21)

## BUGS

**NAME**

speak – word to voice translator

**SYNOPSIS**

**speak** [ –**efpsv** ] [ vocabulary [ output ] ]

**DESCRIPTION**

*Speak* turns a stream of words into utterances and outputs them to a voice synthesizer, or to the specified *output.* It has facilities for maintaining a vocabulary. It receives, from the standard input

– working lines: text of words separated by blanks
– phonetic lines: strings of phonemes for one word preceded and separated by commas. The phonemes may be followed by comma-percent then a ‘replacement part’ – an ASCII string with no spaces. The phonetic code is given in *vs* (V).
– empty lines
– command lines: beginning with **!.** The following command lines are recognized:

| | |
|---|---|
| **!r** file | replace coded vocabulary from file |
| **!w** file | write coded vocabulary on file |
| **!p** | print phonetics for working word |
| **!l** | list vocabulary on standard output with phonetics |
| **!c** word | copy phonetics from working word to specified word |
| **!d** | print decomposition of working word into substrings |
| **!f** *n* | turn off (or on) English preprocessing rule number *n* (see listing for meaning of *n)* |

Each working line replaces its predecessor. Its first word is the ‘working word’. Each phonetic line replaces the phonetics stored for the working word. In particular, a phonetic line of comma only deletes the entry for the working word. Each working line, phonetic line or empty line causes the working line to be uttered. The process terminates at the end of input.

Unknown words are pronounced by rules, and failing that, are spelled. For the builtin part of the rules, see the reference. Spelling is done by taking each character of the word, prefixing it with ‘*’, and looking it up. Unspellable words burp.

Words not found verbatim in the vocabulary are pronounced piecewise. First the word is bracketed by sharps: ‘#...#’. The vocabulary is then searched for the longest fragment that matches the beginning of the word. The phonetic part of the phonetic string is uttered, and the matched fragment is replaced by the replacement part of the phonetic string, if any. The process is repeated until the word is exhausted. A fragment is entered into the vocabulary as a working word prefixed by ‘%’.

*Speak* is initialized with a coded vocabulary stored in file */usr/lib/speak.m.* The vocabulary option substitutes a different file for */usr/lib/speak.m.* Other vocabularies, to be used with option –**e**, exist in /usr/vs/latin.m and /usr/vs/polish.m.

A set of single letter options may appear in any order preceded by **–.** Their meanings are:

| | |
|---|---|
| **e** | suppress English preprocessing |
| **f** | equivalent to ‘f1, f2,...’ |
| **p** | suppress pronunciation by rule |
| **s** | suppress spelling |
| **v** | suppress voice output |

The following input will reconstitute a coded vocabulary, ‘speak.m’, from an ascii listing, ‘speak.v’, that was created using **!l**.

```
(cat speak.v; echo !w speak.m) | speak –v /dev/null
```

**FILES**

/usr/lib/speak.m

-

**SEE ALSO**

M. D. McIlroy, ‘‘Synthetic English Speech by Rule,’’ Computing Science Technical Report #14, Bell Laboratories, 1973
vs (V), vs (IV)

**BUGS**

Excessively long words cause dumps.
Space is not reclaimed from changed entries; use **!w** and **!r** to effect reclamation.
**!p** doesn’t always work as advertised.

**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 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. (Default $n = 100$.)

**p** Make output periodic, i.e. match derivatives at ends. First and last input values should normally agree.

**x** Next 1 (or 2) arguments are lower (and upper) $x$ limits. Normally these limits are calculated from the data. Automatic abcissas start at lower limit (default 0).

**SEE ALSO**

plot (I)

**AUTHOR**

M. D. McIlroy

**BUGS**

A limit of 1000 input points is enforced silently.

## 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 reformatted. The first line after .TS specifies the various columns: it consists of a list of column describers separated by blanks or tabs. Each column describer is a character string made up of the letters ‘n’, ‘r’, ‘c’, ‘l’ and ‘s’, which mean:

| | |
|---|---|
| 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. |

The column describer may be followed by an integer giving the number of spaces between this column and the next; 3 is default. The describer ‘ccr5’ indicates that the first two lines in this column are centered; the third and remaining lines are right-adjusted; and the column should be separated from the column to the right by 5 spaces. Letting \t represent a tab (which must be typed as a genuine tab) the input

```
.TS
cccl sccn sscn
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

```
      Household Population
   Town           Households
               Number     Size
Bedminster         789     3.26
Bernards Twp.     3087     3.74
Bernardsville     2018     3.30
Bound Brook       3425     3.04
Branchburg        1644     3.49
Bridgewater       7897     3.81
Far Hills          240     3.19
```

If no arguments are given, *tbl* reads the standard input, so it may be used as a filter. When it is used with *eqn* or *neqn* the *tbl* command should be first, to minimize the volume of data passed through pipes.

## BUGS

No column describer may end with ‘s’.

## NAME

tmg – compiler-compiler

## SYNOPSIS

**tmg** name

## DESCRIPTION

*Tmg* produces a translator for the language whose parsing and translation rules are described in file *name***.t**. The new translator appears in a.out and may be used thus:

**a.out** input [ output ]

Except in rare cases input must be a randomly addressable file. If no output file is specified, the standard output file is assumed.

## FILES

*name***.s**: assembly language version of *name***.t**  
/usr/lib/tmg: the compiler-compiler  
/usr/lib/tmg[abc], /lib/libs.a: libraries  
alloc.d: scratch file for table storage

## SEE ALSO

A Manual for the Tmg Compiler-writing Language, internal memorandum.

## DIAGNOSTICS

Syntactic errors result in "???" followed by the offending line.  
Situations such as space overflow with which the Tmg processor or a Tmg-produced processor can not cope result in a descriptive comment and a dump.

## AUTHOR

M. D. McIlroy

## BUGS

Footnote 1 of Section 9.2 of Tmg Manual is not enforced, causing trouble.  
Restrictions (7.) against mixing bundling primitives should be lifted.  
Certain hidden reserved words exist: gpar, classtab, trans, goto, alt, salt.  
Octal digits include 8=10 and 9=11.

**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 |

**BUGS**

**NAME**

units − conversion program

**SYNOPSIS**

*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

**BUGS**

**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**

crfork, crexit, crread, crwrite, crexch, crprior – coroutine scheme

**SYNOPSIS**

**int crfork(** [ **stack, nwords** ] **)**
**int stack[];**
**int nwords;**

**crexit()**

**int crread(connector, buffer, nbytes)**
**int *connector[2];**
**char *buffer;**
**int nbytes;**

**crwrite(connector, buffer, nbytes)**
**int *connector[2];**
**char *buffer;**
**int nbytes;**

**crexch(conn1, conn2, i)**
**int *conn1[2], *conn2[2];**
**int i;**

**#define logical char ***
**crprior(p)**
**logical p;**

**DESCRIPTION**

These functions are named by analogy to *fork, exit, read, write* (II). They establish and synchronize ‘coroutines’, which behave in many respects like a set of processes working in the same address space. The functions live in */usr/lib/cr.a.*

Coroutines are placed on queues to indicate their state of readiness. One coroutine is always distinguished as ‘running’. Coroutines that are runnable but not running are registered on a ‘ready queue’. The head member of the ready queue is started whenever no other coroutine is specifically caused to be running.

Each connector heads two queues: *Connector[0]* is the queue of unsatisfied *crreads* outstanding on the connector. *Connector[1]* is the queue of *crwrites.* All queues must start empty, *i.e.* with heads set to zero.

*Crfork* is normally called with no arguments. It places the running coroutine at the head of the ready queue, creates a new coroutine, and starts the new one running. *Crfork* returns immediately in the new coroutine with value 0, and upon restarting of the old coroutine with value 1.

*Crexit* stops the running coroutine and does not place it in any queue.

*Crread* copies characters from the *buffer* of the *crwrite* at the head of the *connector’s* write queue to the *buffer* of *crread.* If the write queue is empty, copying is delayed and the running coroutine is placed on the read queue. The number of characters copied is the minimum of *nbytes* and the number of characters remaining in the write *buffer,* and is returned as the value of *crread.* After copying, the location of the write *buffer* and the corresponding *nbytes* are updated appropriately. If zero characters remain, the coroutine of the *crwrite* is moved to the head of the ready queue. If the write queue remains nonempty, the head member of the read queue is moved to the head of the ready queue.

*Crwrite* queues the running coroutine on the *connector’s* write queue, and records the fact that *nbytes* (zero or more) characters in the string *buffer* are available to *crreads.* If the read queue is not empty, its head member is started running.

*Crexch* exchanges the read queues of connectors *conn1* and *conn2* if *i*=0; and it exchanges the write queues if *i*=1. If a nonempty read queue that had been paired with an empty write queue becomes paired with a nonempty write queue, *crexch* moves the head member of that read queue to the head of the ready queue.

*Crprior* sets a priority on the running coroutine to control the queuing of *crreads* and *crwrites.* When queued, the running coroutine will take its place before coroutines whose priorities exceed its own priority and after others. Priorities are compared as logical, *i.e.* unsigned, quantities. Initially each coroutine's priority is set as large as possible, so default queuing is FIFO.

**Storage allocation.** The old and new coroutine share the same activation record in the function that invoked *crfork*, so only one may return from the invoking function, and then only when the other has completed execution in that function.

The activation record for each function execution is dynamically allocated rather than stacked; a factor of 3 in running time overhead can result if function calls are very frequent. The overhead may be overcome by providing a separate stack for each coroutine and dispensing with dynamic allocation. The base (lowest) address and size of the new coroutine's stack are supplied to *crfork* as optional arguments *stack* and *nwords.* Stacked allocation and dynamic allocation cannot be mixed in one run. For stacked operation, obtain the coroutine functions from */usr/lib/scr.a* instead of */usr/lib/cr.a.*

**FILES**

/usr/lib/cr.a
/usr/lib/scr.a

**DIAGNOSTICS**

'rsave doesn't work' – an old C compilation has called 'rsave'. It must be recompiled to work with the coroutine scheme.

**BUGS**

Under /usr/lib/cr.a each function has just 12 words of anonymous stack for hard expressions and arguments of further calls, regardless of actual need. There is no checking for stack overflow. Under /usr/lib/scr.a stack overflow checking is not rigorous.

## NAME

ms – macros for formatting manuscripts

## SYNOPSIS

**nroff –ms** [ options ] file ...
**troff –ms** [ options ] file ...

## DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers. When producing 2-column output on a terminal, its output should be filtered through *col* (I).

The package supports three different formats: BTL technical memorandum with cover sheet, released paper with cover sheet, and an abbreviated 'debugging' form without cover sheet.

The macro requests are defined in the attached Request Reference. Many *nroff* and *troff* requests are unsafe in conjunction with this package, however the requests listed below may be used with impunity after the first .PP.

| | |
|---|---|
| .bp | begin new page |
| .br | break output line here |
| .sp n | insert n spacing lines |
| .ls n | (line spacing) n=1 single, n=2 double space |
| .na | no alignment of right margin |

Output of the *eqn, neqn* and *tbl* (I) preprocessors for equations and tables is acceptable as input.

## FILES

/usr/lib/tmac.s

## SEE ALSO

eqn (I), nroff (I), troff (I), tbl (VI)

## BUGS

REQUEST REFERENCE

| Request Form | Initial Value | Cause Break | Explanation |
|---|---|---|---|
| .1C | yes | yes | One column format on a new page. |
| .2C | no | yes | Two column format. |
| .AB | no | yes | Begin abstract. |
| .AE | - | yes | End abstract. |
| .AI | no | yes | Author's institution follows. Suppressed in TM. |
| .AU *x y* | no | yes | Author's name follows. *x* is location and *y* is extension, ignored except in TM. |
| .B | no | no | Boldface text follows. |
| .CS *x*... | - | yes | Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references. |
| .DA *x* | nroff | no | 'Date line' at bottom of page is *x*. Default is today. |
| .DE | - | yes | End displayed text. Implies .KE. |
| .DS *x* | no | yes | Start of displayed text, to appear verbatim line-by-line. *x*=I for indented display (default), *x*=L for left-justified on the page, *x*=C for centered. Implies .KS. |
| .EN | - | yes | Space after equation produced by *eqn* or *neqn.* |
| .EQ *x* | - | yes | Space before equation. Equation number is *x*. |
| .FE | - | yes | End footnote. |
| .FS | no | no | Start footnote. The note will be moved to the bottom of the page. |
| .HO | - | no | 'Bell Laboratories, Holmdel, New Jersey 07733'. |
| .I | no | no | Italic text follows. |
| .IP *x y* | no | yes | Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5). |
| .KE | - | yes | End keep. Put kept text on next page if not enough room. |
| .KF | no | yes | Start floating keep. If the kept text must be moved to the next page, float later text back to this page. |
| .KS | no | yes | Start keeping following text. |
| .LG | no | no | Make letters larger. |
| .LP | yes | yes | Start left-blocked paragraph. |
| .MH | - | no | 'Bell Laboratories, Murray Hill, New Jersey 07974'. |
| .ND | troff | no | No date line at bottom of page. |
| .NH *n* | - | yes | Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1). |
| .NL | yes | no | Make letters normal size. |
| .OK | - | yes | 'Other keywords' for TM cover sheet follow. |
| .PP | no | yes | Begin paragraph. First line indented. |
| .R | yes | no | Roman text follows. |
| .RE | - | yes | End relative indent level. |
| .RP | no | - | Cover sheet and first page for released paper. Must precede other requests. |
| .RS | - | yes | Start level of relative indentation. Following .IP's measured from current indentation. |
| .SG *x* | no | yes | Insert signature(s) of author(s), ignored except in TM. *x* is the reference line (initials of author and typist). |
| .SH | - | yes | Section head follows, font automatically bold. |
| .SM | no | no | Make letters smaller. |
| .TL | no | yes | Title follows. |
| .TM *x y z* | no | - | BTL TM cover sheet and first page, *x*=TM number, *y*=(quoted list of) case number(s), *z*=file number. Must precede other requests. |
| .WH | - | no | 'Bell Laboratories, Whippany, New Jersey 07981'. |

## 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)**

**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* (VI) for a description of the meaning of the subroutines.

There are four libraries containing these routines, one that produces general graphics commands on the standard output, and one each for the vt0 storage scope, the Diablo plotting terminal and the Tektronix 4014 terminal. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

## FILES

| | |
|---|---|
| /usr/lib/plot.a | produces output for plotting filters |
| /usr/lib/vt0.a | produces output on vt0 storage scope |
| /usr/lib/gsip.a | produces output on Diablo terminal |
| /usr/lib/tek.a | produces output for the Tektronix 4014 terminal |

## SEE ALSO

plot (VI), graph (VI)

## BUGS

**NAME**

salloc – string allocation and manipulation

**SYNOPSIS**

(get size in r0)
**jsr pc,allocate**
(header address in r1)

(get source header address in r0,
destination header address in r1)
**jsr pc,copy**

**jsr pc,wc**

(all following routines assume r1 contains header address)

**jsr pc,release**

(get character in r0)
**jsr pc,putchar**

**jsr pc,lookchar**
(character in r0)

**jsr pc,getchar**
(character in r0)

(get character in r0)
**jsr pc,alterchar**

(get position in r0)
**jsr pc,seekchar**

**jsr pc,backspace**
(character in r0)

(get word in r0)
**jsr pc,putword**

**jsr pc,lookword**
(word in r0)

**jsr pc,getword**
(word in r0)

(get word in r0)
**jsr pc,alterword**

**jsr pc,backword**
(word in r0)

**jsr pc,length**
(length in r0)

**jsr pc,position**
(position in r0)

**jsr pc,rewind**

**jsr pc,create**

**jsr pc,fsfile**

**jsr pc,zero**

**DESCRIPTION**

This package is a complete set of routines for dealing with almost arbitrary length strings of words and bytes. It lives in */lib/libs.a.* The strings are stored on a disk file, so the sum of their lengths can be considerably larger than the available core. A small buffer cache makes for rea-

-

sonable speed.

For each string there is a header of four words, namely a write pointer, a read pointer and pointers to the beginning and end of the block containing the string. Initially the read and write pointers point to the beginning of the string. All routines that refer to a string require the header address in r1. Unless the string is destroyed by the call, upon return r1 will point to the same string, although the string may have grown to the extent that it had to be be moved.

*Allocate* obtains a string of the requested size and returns a pointer to its header in r1.

*Release* releases a string back to free storage.

*Putchar* and *putword* write a byte or word respectively into the string and advance the write pointer.

*Lookchar* and *lookword* read a byte or word respectively from the string but do not advance the read pointer.

*Getchar* and *getword* read a byte or word respectively from the string and advance the read pointer.

*Alterchar* and *alterword* write a byte or word respectively into the string where the read pointer is pointing and advance the read pointer.

*Backspace* and *backword* read the last byte or word written and decrement the write pointer.

All write operations will automatically get a larger block if the current block is exceeded. All read operations return with the error bit set if attempting to read beyond the write pointer.

*Seekchar* moves the read pointer to the offset specified in r0.

*Length* returns the current length of the string (beginning pointer to write pointer) in r0.

*Position* returns the current offset of the read pointer in r0.

*Rewind* moves the read pointer to the beginning of the string.

*Create* returns the read and write pointers to the beginning of the string.

*Fsfile* moves the read pointer to the current position of the write pointer.

*Zero* zeros the whole string and sets the write pointer to the beginning of the string.

*Copy* copies the string whose header pointer is in r0 to the string whose header pointer is in r1. Care should be taken in using the copy instruction since r1 will be changed if the contents of the source string is bigger than the destination string.

*Wc* forces the contents of the internal buffers and the header blocks to be written on disc.

An in-core version of this allocator exists in *dc* (I), and a permanent-file version exists in *form* and *fed* (VI).

**FILES**

| | |
|---|---|
| /lib/libs.a | library, accessed by *ld ... -ls* |
| alloc.d | temporary file for string storage |

**SEE ALSO**

alloc (III)

**DIAGNOSTICS**

‘error in copy’ – disk write error encountered in *copy.*
‘error in allocator’ – routine called with bad header pointer.
‘cannot open output file’ – temp file *alloc.d* cannot be created or opened.
‘out of space’ – no sufficiently large block or no header is available for a new or growing block.

**BUGS**

**NAME**

ac − login accounting

**SYNOPSIS**

**ac** [ −**w** wtmp ] [ −**p** ] [ −**d** ] people

**DESCRIPTION**

*Ac* produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. −**w** is used to specify an alternate *wtmp* file. −**p** prints individual totals; without this option, only totals are printed. −**d** causes a printout for each midnight to midnight period. Any *people* will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/adm/wtmp* is used.

The accounting file */usr/adm/wtmp* is maintained by *init* and *login.* Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

**FILES**

/usr/adm/wtmp

**SEE ALSO**

init (VIII), login (I), wtmp (V).

**BUGS**

**NAME**

boot procedures – UNIX startup

**DESCRIPTION**

*How to start UNIX.* UNIX is started by placing it in core at location zero and transferring to zero. Since the system is not reenterable, it is necessary to read it in from disk or tape.

The *tp* command places a bootstrap program on the otherwise unused block zero of the tape. The DECtape version of this program is called *tboot,* the magtape version *mboot.* If *tboot* or *mboot* is read into location zero and executed there, it will type '=' on the console, read in a *tp* entry name, load that entry into core, and transfer to zero. Thus one way to run UNIX is to maintain the system code on a tape using *tp.* Caution: the file /usr/mdec/tboot (DECtape) or /usr/mdec/mboot (magtape) must be present when the tape is made! When a boot is required, execute (somehow) a program which reads in and jumps to the first block of the tape. In response to the '=' prompt, type the entry name of the system on the tape (we use plain 'unix'). It is strongly recommended that a current version of the system be maintained in this way, even if it is usually booted from disk.

The standard DEC ROM which loads DECtape is sufficient to read in *tboot,* but the magtape ROM loads block one, not zero. If no suitable ROM is available, magtape and DECtape programs are presented below which may be manually placed in core and executed.

The system can also be booted from a disk file with the aid of the *uboot* program. When read into location 0 and executed, *uboot* reads a single character (either **p** or **k** for RP or RK, both drive 0) to specify which device is to be searched. Then it reads a UNIX pathname from the console, finds the corresponding file on the given device, loads that file into core location zero, and transfers to it. *Uboot* operates under very severe space constraints. It supplies no prompts, except that it echoes a carriage return and line feed after the **p** or **k.** No diagnostic is provided if the indicated file cannot be found, nor is there any means of correcting typographical errors in the file name except to start the program over. If it fails to find the file, however, it jumps back to its start, so another try can be attempted, starting again with the **p** or **k.** Notice that *uboot* will only load a file from drive 0, and the file system it searches must start at the beginning of the disk. *Uboot* itself usually resides in the otherwise unused block 0 of the disk, so it can be loaded by ROM program; *mkfs* can be used to put it there when the file system is created. It can also be loaded from a *tp* tape as described above.

*The switches.* The console switches play an important role in the use and especially the booting of UNIX. During operation, the console switches are examined 60 times per second, and the contents of the address specified by the switches are displayed in the display register. (This is not true on the 11/40 since there is no display register on that machine.) If the switch address is even, the address is interpreted in kernel (system) space; if odd, the rounded-down address is interpreted in the current user space.

If any diagnostics are produced by the system, they are printed on the console only if the switches are non-zero. Thus it is wise to have a non-zero value in the switches at all times.

During the startup of the system, the *init* program (VIII) reads the switches and will come up single-user if the switches are set to 173030.

It is unwise to have a non-existent address in the switches. This causes a bus error in the system (displayed as 177777) at the rate of 60 times per second. If there is a transfer of more than 16ms duration on a device with a data rate faster than the bus error timeout (about 10µs) then a permanent disk non-existent-memory error will occur.

*ROM programs.* Here are some programs which are suitable for installing in read-only memories, or for manual keying into core if no ROM is present. Each program is position-independent but should be placed well above location 0 so it will not be overwritten. Each reads a block from the beginning of a device into core location zero. The octal words constituting the program are listed on the left.

DECtape (drive 0) from endzone:

```
012700          mov     $tcba,r0
177346
010040          mov     r0,–(r0)                / use tc addr for wc
012710          mov     $3,(r0)                 / read bn forward
000003
105710    1:    tstb    (r0)                    / wait for ready
002376          bge     1b
112710          movb    $5,(r0)                 / read (forward)
000005
000777          br      .                       / loop; now halt and start at 0
```

DECtape (drive 0) with search:

```
012700    1:    mov     $tcba,r0
177346
010040          mov     r0,–(r0)                / use tc addr for wc
012740          mov     $4003,–(r0)             / read bn reverse
004003
005710    2:    tst     (r0)
002376          bge     2b                      / wait for error
005760          tst     –2(r0)                  / loop if not end zone
177776
002365          bge     1b
012710          mov     $3,(r0)                 / read bn forward
000003
105710    2:    tstb    (r0)                    / wait for ready
002376          bge     2b
112710          movb    $5,(r0)                 / read (forward)
000005
105710    2:    tstb    (r0)                    / wait for ready
002376          bge     2b
005007          clr     pc                      / transfer to zero
```

Caution: both of these DECtape programs will (literally) blow a fuse if 2 drives are dialed to zero.

Magtape from load point:

```
012700          mov     $mtcma,r0
172526
010040          mov     r0,–(r0)                / usr mt addr for wc
012740          mov     $60003,–(r0)            / read 9–track
060003
000777          br      .                       / loop; now halt and start at 0
```

RK (drive 0):

```
012700          mov     $rkda,r0
177412
005040          clr     –(r0)                   / rkda cleared by start
010040          mov     r0,–(r0)
012740          mov     $5,–(r0)
000005
105710    1:    tstb    (r0)
002376          bge     1b
005007          clr     pc
```

RP (drive 0)

```
012700          mov     $rpmr,r0
176726
005040          clr     –(r0)
005040          clr     –(r0)
005040          clr     –(r0)
010040          mov     r0,–(r0)
012740          mov     $5,–(r0)
000005
105710    1:    tstb    (r0)
002376          bge     1b
005007          clr     pc
```

**FILES**

/unix – UNIX code
/usr/mdec/mboot – *tp* magtape bootstrap
/usr/mdec/tboot – *tp* DECtape bootstrap
/usr/mdec/uboot – file system bootstrap

**SEE ALSO**

tp (I), init (VIII)

**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.

Only the super-user is allowed to change the group of a file, in order to simplify as yet unimplemented accounting procedures.

**SEE ALSO**

chown (VIII)

**FILES**

/etc/group

**BUGS**

**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.

Only the super-user is allowed to change the owner of a file, in order to simplify as yet unimplemented accounting procedures.

**FILES**

/etc/passwd

**SEE ALSO**

chgrp (VIII)

**BUGS**

**NAME**

clri – clear i-node

**SYNOPSIS**

**clri** i-number [ filesystem ]

**DESCRIPTION**

*Clri* writes zeros on the 32 bytes occupied by the i-node numbered *i-number.* If the *file system* argument is given, the i-node resides on the given device, otherwise on a default file system. The file system argument must be a special file name referring to a device containing a file system. After *clri,* any blocks in the affected file will show up as ‘‘missing’’ in an *icheck* of of the file system.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

**BUGS**

Whatever the default file system is, it is likely to be wrong. Specify the file system explicitly.

If the file is open, *clri* is likely to be ineffective.

-

**NAME**

crash – what to do when the system crashes

**DESCRIPTION**

This section gives at least a few clues about how to proceed if the system crashes. It can't pretend to be complete.

*How to bring it back up.* If the reason for the crash is not evident (see below for guidance on 'evident') you may want to try to dump the system if you feel up to debugging. At the moment a dump can be taken only on magtape. With a tape mounted and ready, stop the machine, load address 44, and start. This should write a copy of all of core on the tape with an EOF mark. Caution: Any error is taken to mean the end of core has been reached. This means that you must be sure the ring is in, the tape is ready, and the tape is clean and new. If the dump fails, you can try again, but some of the registers will be lost. See below for what to do with the tape.

In restarting after a crash, always bring up the system single-user. This is accomplished by following the directions in *boot procedures* (VIII) as modified for your particular installation; a single-user system is indicated by having a particular value in the switches (173030 unless you've changed *init)* as the system starts executing. When it is running, perform a *dcheck* and *icheck* (VIII) on all file systems which could have been in use at the time of the crash. If any serious file system problems are found, they should be repaired. When you are satisfied with the health of your disks, check and set the date if necessary, then come up multi-user. This is most easily accomplished by changing the single-user value in the switches to something else, then logging out by typing an EOT.

To even boot UNIX at all, three files (and the directories leading to them) must be intact. First, the initialization program */etc/init* must be present and executable. If it is not, the CPU will loop in user mode at location 6. For *init* to work correctly, */dev/tty8* and */bin/sh* must be present. If either does not exist, the symptom is best described as thrashing. *Init* will go into a *fork/exec* loop trying to create a Shell with proper standard input and output.

If you cannot get the system to boot, a runnable system must be obtained from a backup medium. The root file system may then be doctored as a mounted file system as described below. If there are any problems with the root file system, it is probably prudent to go to a backup system to avoid working on a mounted file system.

*Repairing disks.* The first rule to keep in mind is that an addled disk should be treated gently; it shouldn't be mounted unless necessary, and if it is very valuable yet in quite bad shape, perhaps it should be dumped before trying surgery on it. This is an area where experience and informed courage count for much.

The problems reported by *icheck* typically fall into two kinds. There can be problems with the free list: duplicates in the free list, or free blocks also in files. These can be cured easily with an *icheck –s.* If the same block appears in more than one file or if a file contains bad blocks, the files should be deleted, and the free list reconstructed. The best way to delete such a file is to use *clri* (VIII), then remove its directory entries. If any of the affected files is really precious, you can try to copy it to another device first.

*Dcheck* may report files which have more directory entries than links. Such situations are potentially dangerous; *clri* discusses a special case of the problem. All the directory entries for the file should be removed. If on the other hand there are more links than directory entries, there is no danger of spreading infection, but merely some disk space that is lost for use. It is sufficient to copy the file (if it has any entries and is useful) then use *clri* on its inode and remove any directory entries that do exist.

Finally, there may be inodes reported by *dcheck* that have 0 links and 0 entries. These occur on the root device when the system is stopped with pipes open, and on other file systems when the system stops with files that have been deleted while still open. A *clri* will free the inode, and an *icheck -s* will recover any missing blocks.

*Why did it crash?* UNIX types a message on the console typewriter when it voluntarily crashes. Here is the current list of such messages, with enough information to provide a hope at least of the remedy. The message has the form 'panic: ...', possibly accompanied by other information. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

blkdev
: The *getblk* routine was called with a nonexistent major device as argument. Definitely hardware or software error.

devtab
: Null device table entry for the major device used as argument to *getblk.* Definitely hardware or software error.

iinit
: An I/O error reading the super-block for the root file system during initialization.

out of inodes
: A mounted file system has no more i-nodes when creating a file. Sorry, the device isn't available; the *icheck* should tell you.

no fs
: A device has disappeared from the mounted-device table. Definitely hardware or software error.

no imt
: Like 'no fs', but produced elsewhere.

no inodes
: The in-core inode table is full. Try increasing NINODE in param.h. Shouldn't be a panic, just a user error.

no clock
: During initialization, neither the line nor programmable clock was found to exist.

swap error
: An unrecoverable I/O error during a swap. Really shouldn't be a panic, but it is hard to fix.

unlink − iget
: The directory containing a file being deleted can't be found. Hardware or software.

out of swap space
: A program needs to be swapped out, and there is no more swap space. It has to be increased. This really shouldn't be a panic, but there is no easy fix.

out of text
: A pure procedure program is being executed, and the table for such things is full. This shouldn't be a panic.

trap
: An unexpected trap has occurred within the system. This is accompanied by three numbers: a 'ka6', which is the contents of the segmentation register for the area in which the system's stack is kept; 'aps', which is the location where the hardware stored the program status word during the trap; and a 'trap type' which encodes which trap occurred. The trap types are:

| | |
|---|---|
| 0 | bus error |
| 1 | illegal instruction |
| 2 | BPT/trace |
| 3 | IOT |
| 4 | power fail |
| 5 | EMT |
| 6 | recursive system call (TRAP instruction) |
| 7 | 11/70 cache parity, or programmed interrupt |

10 floating point trap
11 segmentation violation

In some of these cases it is possible for octal 20 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred. If you wish to examine the stack after such a trap, either dump the system, or use the console switches to examine core; the required address mapping is described below.

*Interpreting dumps.* All file system problems should be taken care of before attempting to look at dumps. The dump should be read into the file */usr/sys/core; cp* (I) will do. At this point, you should execute *ps −alxk* and *who* to print the process table and the users who were on at the time of the crash. You should dump ( *od* (I)) the first 30 bytes of */usr/sys/core.* Starting at location 4, the registers R0, R1, R2, R3, R4, R5, SP and KDSA6 (KISA6 for 11/40s) are stored. If the dump had to be restarted, R0 will not be correct. Next, take the value of KA6 (location 22(8) in the dump) multiplied by 100(8) and dump 1000(8) bytes starting from there. This is the per-process data associated with the process running at the time of the crash. Relabel the addresses 140000 to 141776. R5 is C's frame or display pointer. Stored at (R5) is the old R5 pointing to the previous stack frame. At (R5)+2 is the saved PC of the calling procedure. Trace this calling chain until you obtain an R5 value of 141756, which is where the user's R5 is stored. If the chain is broken, you have to look for a plausible R5, PC pair and continue from there. Each PC should be looked up in the system's name list using *db* (I) and its ':' command, to get a reverse calling order. In most cases this procedure will give an idea of what is wrong. A more complete discussion of system debugging is impossible here.

**SEE ALSO**

clri, icheck, dcheck, boot procedures (VIII)

**BUGS**

**NAME**

cron – clock daemon

**SYNOPSIS**

**/etc/cron**

**DESCRIPTION**

*Cron* executes commands at specified dates and times according to the instructions in the file /usr/lib/crontab. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file /etc/rc; see *init* (VIII).

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by *cron* every hour. Thus it could take up to an hour for entries to become effective. If it receives a hangup signal, however, the table is examined immediately; so ‘kill −1 ...’ can be used.

**FILES**

/usr/lib/crontab

**SEE ALSO**

init(VIII), sh(I), kill (I)

**DIAGNOSTICS**

None − illegal lines in crontab are ignored.

**BUGS**

A more efficient algorithm could be used. The overhead in running *cron* is about one percent of the machine, exclusive of any commands executed.

## NAME

dcheck – file system directory consistency check

## SYNOPSIS

**dcheck** [ **–i** numbers ] [ filesystem ]

## DESCRIPTION

*Dcheck* reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The **–i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

## FILES

Currently, /dev/rrk2 and /dev/rrp0 are the default file systems.

## DIAGNOSTICS

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

## SEE ALSO

icheck (VIII), fs (V), clri (VIII), ncheck (VIII)

## BUGS

Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

**NAME**

df – disk free

**SYNOPSIS**

**df** [ filesystem ]

**DESCRIPTION**

*Df* prints out the number of free blocks available on a file system. If the file system is unspecified, the free space on all of the normally mounted file systems is printed.

**FILES**

/dev/rf?, /dev/rk?, /dev/rp?

**SEE ALSO**

icheck (VIII)

**BUGS**

-

**NAME**

dpd – data phone daemon

**SYNOPSIS**

**/etc/dpd**

**DESCRIPTION**

*Dpd* is the 201 data phone daemon. It is designed to submit jobs to the Honeywell 6070 computer via the GRTS interface.

*Dpd* uses the directory */usr/dpd.* The file *lock* in that directory is used to prevent two daemons from becoming active. After the daemon has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with **df.** Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

**S** directs *dpd* to generate a unique snumb card. This card is generated by incrementing the first word of the file */usr/dpd/snumb* and converting that to three-digit octal concatenated with the station ID.

**L** specifies that the remainder of the line is to be sent as a literal.

**B** specifies that the rest of the line is a file name. That file is to be sent as binary cards.

**F** is the same as **B** except a form feed is prepended to the file.

**U** specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.

**M** is followed by a user ID; after the job is sent, the snumb number and the first line of information in the file is mailed to the user to verify the sending of the job.

Any error encountered will cause the daemon to drop the call, wait up to 20 minutes and start over. This means that an improperly constructed *df* file may cause the same job to be submitted every 20 minutes.

While waiting, the daemon checks to see that the *lock* file still exists. If it is gone, the daemon will exit.

**FILES**

/dev/dn0, /dev/dp0, /usr/dpd/*

**SEE ALSO**

opr (I)

## NAME

dump – incremental file system dump

## SYNOPSIS

**dump** [ key [ arguments ] filesystem ]

## DESCRIPTION

*Dump* makes an incremental file system dump on magtape of all files changed after a certain date. The *key* argument specifies the date and other options about the dump. *Key* consists of characters from the set **abcfiu0hds.**

**a** Normally files larger than 1000 blocks are not incrementally dump; this flag forces them to be dumped.

**b** The next argument is taken to be the maximum size of the dump tape in blocks (see **s**).

**c** If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)

**f** Place the dump on the next argument file instead of the tape.

**i** the dump date is taken from the entry in the file /etc/dtab corresponding to the last time this file system was dumped with the **-u** option.

**u** the date just prior to this dump is written on /etc/dtab upon successful completion of this dump. This file contains a date for every file system dumped with this option.

**0** the dump date is taken as the epoch (beginning of time). Thus this option causes an entire file system dump to be taken.

**h** the dump date is some number of hours before the current date. The number of hours is taken from the next argument in *arguments.*

**d** the dump date is some number of days before the current date. The number of days is taken from the next argument in *arguments.*

**s** the size of the dump tape is specified in feet. The number of feet is taken from the next argument in *arguments.* It is assumed that there are 9 standard UNIX records per foot. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2200 feet.

If no arguments are given, the *key* is assumed to be **i** and the file system is assumed to be /dev/rp0.

Full dumps should be taken on quiet file systems as follows:

```
dump 0u /dev/rp0
ncheck /dev/rp0
```

The *ncheck* will come in handy in case it is necessary to restore individual files from this dump. Incremental dumps should then be taken when desired by:

```
dump
```

When the incremental dumps get cumbersome, a new complete dump should be taken. In this way, a restore requires loading of the complete dump tape and only the latest incremental tape.

## DIAGNOSTICS

If the dump requires more than one tape, it will ask you to change tapes. Reply with a new-line when this has been done. If the first block on the new tape is not writable, e.g. because you forgot the write ring, you get a chance to fix it. Generally, however, read or write failures are fatal.

## FILES

/dev/mt0magtape
/dev/rp0 default file system
/etc/dtab

-

**SEE ALSO**

restor (VIII), ncheck (VIII), dump (V)

**BUGS**

**NAME**

getty – set typewriter mode

**SYNOPSIS**

**/etc/getty** [ char ]

**DESCRIPTION**

*Getty* is invoked by *init* (VIII) immediately after a typewriter is opened following a dial-up. It reads the user's name and invokes the *login* command (I) with the name as argument. While reading the name *getty* attempts to adapt the system to the speed and type of terminal being used.

*Init* calls *getty* with an argument specified by the *ttys* file entry for the typewriter line. Arguments other than '0' can be used to make *getty* treat the line specially. Normally, it sets the speed of the interface to 300 baud, specifies that raw mode is to be used (break on every character), that echo is to be suppressed, and either parity allowed. It types the ‘‘login:’’ message, which includes the characters which put the Terminet 300 terminal into full-duplex and return the GSI terminal to non-graphic mode. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the ‘‘break’’ (‘‘interrupt’’) key. The speed is then changed to 150 baud and the ‘‘login:’’ is typed again, this time including the character sequence which puts a Teletype 37 into full-duplex. If a subsequent null character is received, the speed is changed back to 300 baud.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *stty* (II)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, login is called with the user's name as argument.

**SEE ALSO**

init (VIII), login (I), stty (II), ttys (V)

**BUGS**

**NAME**

glob – generate command arguments

**SYNOPSIS**

**/etc/glob** command [ arguments ]

**DESCRIPTION**

*Glob* is used to expand arguments to the shell containing ‘‘*’’, ‘‘[’’, or ‘‘?’’. It is passed the argument list containing the metacharacters; *glob* expands the list and calls the indicated command. The actions of *glob* are detailed in the Shell writeup.

**SEE**

sh (I)

**BUGS**

## NAME

icheck – file system storage consistency check

## SYNOPSIS

**icheck** [ **–s** ] [ **–b** numbers ] [ filesystem ]

## DESCRIPTION

*Icheck* examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. If the file system is not specified, a set of default file systems is checked. The normal output of *icheck* includes a report of

The number of blocks missing; i.e. not in any file nor in the free list,
The number of special files,
The total number of files,
The number of large and huge files,
The number of directories,
The number of indirect blocks, and the number of double-indirect blocks in huge files,
The number of blocks used in files,
The number of free blocks.

The **–s** flag causes *icheck* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The **–s** flag causes the normal output reports to be suppressed.

Following the **–b** flag is a list of block numbers; whenever any of the named blocks turns up in a file, a diagnostic is produced.

*Icheck* is faster if the raw version of the special file is used, since it reads the i-list many blocks at a time.

## FILES

Currently, /dev/rrk2 and /dev/rrp0 are the default file systems.

## SEE ALSO

dcheck (VIII), ncheck (VIII), fs (V), clri (VIII), restor(VIII)

## DIAGNOSTICS

For duplicate blocks and bad blocks (which lie outside the file system) *icheck* announces the difficulty, the i-number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and *icheck* considers it to contain 0. ‘‘Bad freeblock’’ means that a block number outside the available space was encountered in the free list. ‘‘*n* dups in free’’ means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

## BUGS

Since *icheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.
It believes even preposterous super-blocks and consequently can get core images.

## NAME

init – process control initialization

## SYNOPSIS

**/etc/init**

## DESCRIPTION

*Init* is invoked inside UNIX as the last step in the boot procedure. Generally its role is to create a process for each typewriter on which a user may log in.

First, *init* checks to see if the console switches contain 173030. (This number is likely to vary between systems.) If so, the console typewriter **/dev/tty8** is opened for reading and writing and the Shell is invoked immediately. This feature is used to bring up a single-user system. When the system is brought up in this way, the *getty* and *login* routines mentioned below and described elsewhere are not used. If the Shell terminates, *init* starts over looking for the console switch setting.

Otherwise, *init* invokes a Shell, with input taken from the file */etc/rc.* This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then *init* reads the file */etc/ttys* and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0 and 1, the standard input and output. Opening the typewriter will usually involve a delay, since the *open* is not completed until someone is dialed up and carrier established on the channel. Then */etc/getty* is called with argument as specified by the last character of the *ttys* file line. *Getty* reads the user's name and invokes *login* (q.v.) to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. Then the appropriate typewriter is reopened and *getty* is reinvoked.

*Init* catches the *hangup* signal (signal #1) and interprets it to mean that the switches should be examined as in a reboot: if they indicate a multi-user system, the */etc/ttys* file is read again. The Shell process on each line which used to be active in *ttys* but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the *ttys* file and sending a *hangup* signal to the *init* process: use ‘‘kill –1 1.’’

## FILES

/dev/tty?, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

## SEE ALSO

login (I), kill (I), sh (I), ttys (V), getty (VIII)

**NAME**

lpd – line printer daemon

**SYNOPSIS**

**/etc/lpd**

**DESCRIPTION**

*Lpd* is the line printer daemon (spool area handler) invoked by *opr*. It uses the directory */usr/lpd.* The file *lock* in that directory is used to prevent two daemons from becoming active simultaneously. After the daemon has successfully set the lock, it scans the directory for files beginning with ‘‘df.’’ Lines in each *df* file specify files to be printed in the same way as is done by the data-phone daemon dpd (VIII).

**FILES**

| | |
|---|---|
| /usr/lpd/* | spool area |
| /dev/lp | printer |

**SEE ALSO**

dpd (VIII), opr (I)

**BUGS**

## NAME

mkfs − construct a file system

## SYNOPSIS

**/etc/mkfs** special proto

## DESCRIPTION

*Mkfs* constructs a file system by writing on the special file *special* according to the directions found in the prototype file *proto.* The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program (see boot procedures (VIII)). The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the i-list size in blocks (remember there are 16 i-nodes per block). The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **−bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or − to specify set-user-id mode or not. The third is **g** or − for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod* (I)).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, *mkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

If the prototype file cannot be opened and its name consists of a string of digits, *mkfs* builds a file system with a single empty directory on it. The size of the file system is the value of *proto* interpreted as a decimal number. The i-list size is the file system size divided by 43 plus the size divided by 1000. (This corresponds to an average size of three blocks per file for a 4000 block file system and six blocks per file at 40,000.) The boot program is left uninitialized.

A sample prototype specification follows:

```
/usr/mdec/uboot
4872 55
d−−777 3 1
usr     d−−777 3 1
        sh      −−−755 3 1 /bin/sh
        ken     d−−755 6 1
                $
        b0      b−−644 3 1 0 0
        c0      c−−644 3 1 0 0
        $
$
```

## SEE ALSO

file system (V), directory (V), boot procedures (VIII)

## BUGS

It is not possible to initialize a file larger than 64K bytes.
The size of the file system is restricted to 64K blocks.
There should be some way to specify links.

-

## NAME

mknod – build special file

## SYNOPSIS

**/etc/mknod** name [ **c** ] [ **b** ] major minor

## DESCRIPTION

*Mknod* makes a directory entry and corresponding i-node for a special file. The first argument is the *name* of the entry. The second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number).

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file *conf.c.*

## SEE ALSO

mknod (II)

## BUGS

**NAME**

mount – mount file system

**SYNOPSIS**

**/etc/mount** special file [ **–r** ]

**DESCRIPTION**

*Mount* announces to the system that a removable file system is present on the device corresponding to special file *special* (which must refer to a disk or possibly DECtape). The *file* must exist already; it becomes the name of the root of the newly mounted file system.

*Mount* maintains a table of mounted devices; if invoked without an argument it prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

**SEE ALSO**

mount (II), mtab (V), umount (VIII)

**BUGS**

Mounting file systems full of garbage will crash the system.

**NAME**

ncheck – generate names from i-numbers

**SYNOPSIS**

**ncheck** [ –**i** numbers ] [ –**a** ] [ filesystem ]

**DESCRIPTION**

*Ncheck* with no argument generates a pathname vs. i-number list of all files on a set of default file systems. The –**i** flag reduces the report to only those files whose i-numbers follow. –**a** allows printing of the names '.' and '..', which are ordinarily suppressed. A file system may be specified.

The full report is in no useful order, and probably should be sorted.

**SEE ALSO**

dcheck (VIII), icheck (VIII), sort (I)

**BUGS**

## NAME

restor – incremental file system restore

## SYNOPSIS

**restor** key [ arguments ]

## DESCRIPTION

*Restor* is used to read magtapes dumped with the *dump* command. The *key* argument specifies what is to be done. *Key* is a character from the set **trxw.**

**t** The date that the tape was made and the date that was specified in the *dump* command are printed. A list of all of the i-numbers on the tape is also given.

**r** The tape is read and loaded into the file system specified in *arguments.* This should not be done lightly (see below).

**x** Each file on the tape is individually extracted into a file whose name is the file's i-number. If there are *arguments*, they are interpreted as i-numbers and only they are extracted.

**c** If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)

**f** Read the dump from the next argument file instead of the tape.

**i** All read and checksum errors are reported, but will not cause termination.

**w** In conjunction with the **x** option, before each file is extracted, its i-number is typed out. To extract this file, you must respond with **y.**

The **x** option is used to retrieve individual files. If the i-number of the desired file is not known, it can be discovered by following the file system directory search algorithm. First retrieve the *root* directory whose i-number is 1. List this file with *ls −fi 1.* This will give names and i-numbers of sub-directories. Iterating, any file may be retrieved.

The **r** option should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape onto this. Thus

```
/etc/mkfs /dev/rp0 40600
restor r /dev/rp0
```

is a typical sequence to restore a complete dump. Another *restor* can be done to get an incremental dump in on top of this.

A *dump* followed by a *mkfs* and a *restor* is used to change the size of a file system.

## FILES

/dev/mt0

## SEE ALSO

ls (I), dump (VIII), mkfs (VIII), clri (VIII)

## DIAGNOSTICS

There are various diagnostics involved with reading the tape and writing the disk. There are also diagnostics if the i-list or the free list of the file system is not large enough to hold the dump.

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

## BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *restor's* approach is to exit if anything is wrong.

-

**NAME**

sa − Shell accounting

**SYNOPSIS**

sa [ **−abcjlnrstuv** ] [ file ]

**DESCRIPTION**

When a user logs in, if the Shell is able to open the file */usr/adm/sha,* then as each command completes the Shell writes at the end of this file the name of the command, the user, system, and real time consumed, and the user ID. *Sa* reports on, cleans up, and generally maintains this and other accounting files. To turn accounting on and off, the accounting file must be created or destroyed externally.

*Sa* is able to condense the information in */usr/adm/sha* into a summary file */usr/adm/sht* which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system *sha* can grow by 100 blocks per day. The summary file is read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; *sha* is the default. There are zillions of options:

**a** Place all command names containing unprintable characters and those used only once under the name ‘‘***other.’’

**b** Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.

**c** Besides total user, system, and real time for each command print percentage of total time over all commands.

**j** Instead of total minutes time for each category, give seconds per call.

**l** Separate system and user time; normally they are combined.

**n** Sort by number of calls.

**r** Reverse order of sort.

**s** Merge accounting file into summary file */usr/adm/sht* when done.

**t** For each command report ratio of real time to the sum of user and system times.

**u** Superseding all other flags, print for each command in the accounting file the day of the year, time, day of the week, user ID and command name.

**v** If the next character is a digit *n*, then type the name of each command used *n* times or fewer. Await a reply from the typewriter; if it begins with ‘‘y’’, add the command to the category ‘‘**junk**.’’ This is used to strip out garbage.

**FILES**

| | |
|---|---|
| /usr/adm/sha | accounting |
| /usr/adm/sht | summary |

**SEE ALSO**

ac (VIII)

**BUGS**

**NAME**

su – become privileged user

**SYNOPSIS**

**su**

**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 UID set to that of the super-user. To restore normal UID 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 ‘%’.

**SEE ALSO**

sh (I)

**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)

**BUGS**

**NAME**

umount – dismount file system

**SYNOPSIS**

**/etc/umount** special

**DESCRIPTION**

*Umount* announces to the system that the removable file system previously mounted on special file *special* is to be removed.

**SEE ALSO**

mount (VIII), umount (II), mtab (V)

**FILES**

| | |
|---|---|
| /etc/mtab | mounted device table |

**DIAGNOSTICS**

It complains if the special file is not mounted or if it is busy. The file system is busy if there is an open file on it or if someone has his current directory there.

**BUGS**

**NAME**

update – periodically update the super block

**SYNOPSIS**

**update**

**DESCRIPTION**

*Update* is a program that executes the *sync* primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file. See sync (II) for details.

**SEE ALSO**

sync (II), init (VIII)

**BUGS**

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync* temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.

**NAME**

wall – write to all users

**SYNOPSIS**

**/etc/wall**

**DESCRIPTION**

*Wall* reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by ‘‘Broadcast Message ...’’. It is used to warn all users, typically prior to shutting down the system.

The sender should be super-user to override any protections the users may have invoked.

**FILES**

/dev/tty?

**SEE ALSO**

mesg (I), write (I)

**DIAGNOSTICS**

‘‘Cannot send to ...’’ when the open on a user’s tty file fails.

**BUGS**