UNIX™

for the

68000

VOLUME I
The User's Manual

8/23/82

UniSoft
CORPORATION
2405 Fourth Street • Berkeley, CA 94710

PREFACE

to the UniSoft Edition

While updating this documentation for use with UniSoft's UNIX for the 68000, we added examples to the Commands in Volume I, Section 1, and clarified descriptive material where necessary. We are indebted to the many writers who have built up the UNIX documentation over the years, and our intent has been to enhance, rather than to replace, their work. Particular thanks are due to Jeff Schriebman and Asa Romberger, who showed great flexibility in switching from porting to proofreading on short notice, and without whose advice and assistance this revision would not have been possible.

UniSoft Corporation
February 25, 1982

PREFACE

to the University of California Edition

This edition of the manual, while heavily based on the original from Bell Labs, incorporates documentation reflecting the version of UNIX currently running on the Berkeley campus of the University of California. I would like to give special thanks to Vance Vaughan, Roberta Allsman, Dick Peters, Kirk Thege, Jeff Schriebman, and Bill Joy for their help in preparing this edition.

E.M. Gould

PREFACE

to the Seventh Edition

Although this Seventh Edition no longer bears their byline, Ken Thompson and Dennis Ritchie remain the fathers and preceptors of the UNIX time-sharing system. Many of the improvements here described bear their mark. Among many, many other people who have contributed to the further flowering of UNIX, we wish especially to acknowledge the contributions of A. V. Aho, S. R. Bourne, L. L. Cherry, G. L. Chesson, S. I. Feldman, C. B. Haley, R. C. Haight, S. C. Johnson, M. E. Lesk, T. L. Lyon, L. E. McMahon, R. Morris, R. Muha, D. A. Nowitz, L. Wehr, and P. J. Weinberger. We appreciate also the effective advice and criticism of T. A. Dolotta, A. G. Fraser, J. F. Maranzano, and J. R. Mashey; and we remember the important work of the late Joseph F. Ossanna.

B. W. Kernighan
M. D. McIlroy

Introduction to UniSoft UNIX on the 68000

UniSoft Company Profile

UniSoft Corporation was formed in 1981 to provide the UNIX* operating system to OEM's (original equipment manufacturers) of computers, who would in turn supply UNIX to end users.

UNIX is a general purpose interactive operating system originally developed for use on Digital Equipment Corporation (DEC) minicomputers. UniSoft has modified UNIX to run on state of the art microcomputers such as the Motorola 68000. UNIX provides systems programming development and text processing facilities which substantially augment the computing power and flexibility of these computers. UniSoft believes that UNIX will become the standard operating system for all 16 bit and 32 bit computers.

UNIX for the 68000 was chosen as UniSoft's initial product after a market survey and a careful study of the technical problems. The 68000 is a chip with 32 bit internal and 16 bit external addressing which is being used for many of the newer microcomputer systems because of its speed, power, and flexibility.

History of UNIX

The UNIX operating system has finally emerged from its sheltered academic environment and become available commercially at an affordable price. Since it has been lovingly groomed by researchers, professors, and students in hundreds of educational institutions (not to mention Bell Labs, one of the world's largest research facilities) UNIX represents a large, complex, and fairly stable set of programs.

UNIX was originally developed at Bell Labs in 1969 on what was then considered a rather "small" computer, the DEC PDP-7. Two programmers in the Computing Science Research Group, Ken Thompson and Dennis Ritchie, wrote UNIX because the operating systems that were available at that time did not provide the type of programming environment that they wanted.

Unlike many other operating systems overloaded with unnecessary features and fraught with hazards for the unwary, UNIX provides a simple, minimal set of tools (and tools to make tools) for software development and document preparation.

---

*UNIX is a Trademark of Bell Laboratories.

In a short period of time, UNIX became very popular with Bell programmers and computer science researchers, and is now the standard operating system on hundreds of computers throughout the Bell network.

UNIX has also been installed on thousands of other systems, particularly those in colleges and universities. Because of the merits of UNIX as a multi-user programming environment and because Bell made it widely available to educational institutions, UNIX has become one of the major computer science teaching systems. By 1981, there were more than 1700 installations of UNIX in colleges and universities. Jean Yates, co-author (with Rebecca Thomas) of A User Guide to the UNIX System, estimates that over 90% of computer science departments in universities use UNIX systems.

Over the years, UNIX has gone through several revisions. Until recently, the latest version of UNIX available from Bell was Version 7. However, some regional variations also existed. A group within Bell had developed a set of tools called Programmers Workbench (PWB), and the University of California at Berkeley had made several substantial enhancements (referred to collectively as "Berkeley UNIX") to the "standard" UNIX system.

A new UNIX release was announced in November, 1981, in order to provide a more comprehensive and fully standard version of UNIX and to consolidate computer-related goods and services under A.T.&T. This release, System III, integrates all the different versions, eliminates a few programs, and makes available from Bell most of the PWB and the UC Berkeley enhancements. Thus, although System IV and System V are already looming in the realm of rumor, System III currently represents the minimum standard UNIX system.

Even more significantly, Bell's licensing fee structure has also changed, so that for the first time UNIX can be licensed at a price that makes it commercially viable on microcomputers. This now puts the UNIX programming and text processing tools in the hands of small businesses and private users for the first time.

Although UNIX has been thoroughly shaken down over the years of its use in a research environment, it is not now and has never been a system designed primarily for use by non-technical people. That is, UNIX is somewhat less "user-friendly" than a system developed specifically for use by businesses or at home. However, thousands of non-technical people have learned to know and enjoy UNIX, and computer terminals in, for example, university offices are in continuous use by non-academic personnel.

The real value of UNIX lies in its hundreds of utility programs. No other operating system has such a large and powerful set of program development and text processing tools. UNIX provides tools or a means of making tools for almost any application, once you know where to look and what to do when you get there.

In the past, most UNIX users have learned the system by oral tradition. In a university, this is no problem -- there's always someone to ask. However, if you don't have an experienced UNIX user at your elbow, learning by trial and error can be frustrating.

Therefore, this "Introduction to the Introduction" is designed as a brief guide to the most useful commands for maneuvering in UNIX, and as a guide to the UNIX documentation. The three volumes may seem unwieldy, but even at this size they have been distilled from the four volumes that come in, for example, the U.C. Berkeley distribution.

About the UNIX Manuals

UniSoft's edition of the UNIX documentation attempts not only to remove documents which are outdated or which do not apply to UniSoft UNIX for the 68000, but also to present the documentation in a logical sequence.

The first volume is The User's Manual, Volume I. This volume contains brief descriptions of each of the major commands, subroutines, system calls, etc., that can be used or accessed by the average user.

Section 1 of The User's Manual, "Commands", represents a set of programs that can be directly used by all users. As such, Section 1 is the section people use most.

Volumes II and III divide the UNIX world into programming (Volume II) and text processing functions (Volume III). In each volume, there is a progression from non-technical or tutorial documents to more technical and abstract articles about more complex facilities.

Getting Started

The beginning user should start with Volume III. This volume contains "An Introduction to UNIX" and other entry-level documents. Volume III also concentrates on text processing, which is a good way to get practice on UNIX and to learn its features. This Introduction plus the Introduction to Volume I, should give you enough information to get started. Then the tutorials and exercises at the beginning of Volume III will give you more details.

Text Processing

In addition to introducing the UNIX operating system in a tutorial way, Volume III also contains essays and tutorials on text processing and document preparation programs.

UNIX provides several editors, but the line editor _ex_ and its screen-oriented version _vi_ are the most commonly used. Document formatting capability is provided by _nroff_ and _troff_, which produces typeset for printing. The formatting programs are simplified by "macro" packages such as the _ms_ macros, which provide a standard set of commands for standard formatting operations.

Documents can be revised _en masse_, with programs such as the stream editor _sed_ or the transliteration program _tr_. Finally, textual analysis programs such as _awk_ and _lex_ permit editing "scripts" to be written to perform a series of operations on documents.

Program Development

Volume II contains documents on the C programming language (in which UNIX is written) and other program development tools. UNIX is particularly rich in systems programming tools.

In addition to the C language interface, which is obviously well developed, UNIX supports other programming languages.+ The program development tools (which can often be used on text files as well as files of code) enable mass revision of files, close tracking of revisions, archiving, and other resource management functions.

Sandy Emerson
UniSoft Corporation

February 25, 1982

+ UniSoft provides interfaces to FORTRAN, Pascal, and other languages through cross-licensing agreements. The languages and manuals for them may be obtained from the manufacturer of your UniSoft UNIX system.

The All-Purpose Rudimentary Users' Guide to UNIX

The following should give you, in very concise form, enough information to begin to find your way around in UNIX. The chart form is designed to supplement the clouds of fine print that have gathered around UNIX operations over the years; however, many details are omitted. You will need the User's Manual and the appropriate supplementary documentation in order to move up from Sunday driver to UNIX speedster.

HOW TO	COMMANDS
LOG IN	Boot system up and type Control-D to the single-user "(#)" prompt, then respond to login: with your user name and a carriage return. (Commands are always sent to the system with a carriage return).
CREATE A FILE	ex <filename> create a file by editing. Give the file a name and add text to it by typing 'a' to the colon (:) prompt. Many commands also open a new file automatically, when a new name is given for the new file. For example, "copy": cp oldname newname will create newname automatically and copy oldname into it.
MAKE A DIRECTORY	mkdir <directory name> give the directory a name. To use this directory and add files to it, use: cd <directory name> to "change directory" to the new one. Directories exist in a tree structure. Directories have parents and children, starting with the single "root" directory which is the parent of all the other directories.
CHANGE DIRECTORY	cd to the directory <name>. To go up one level, use: cd .. ".." is the parent of the directory you are in. In this way you

can climb up and down directory "trees" to examine the contents of the system without having to know specific directory names in advance.

LIST DIRECTORY CONTENTS

ls
to see the names of files and directories. To see the permissions on various files, type:
ls -l
(That's "l" as in "long", not the number "1"). "Read, write, execute" (rwx) permissions go (from left to right), owner: group: public. If you are not the owner of a file then you must have at least "read" permission as a member of "group" or "public" in order to access and/or move the file. "d" at the beginning of the permission string indicates a directory.

FIND WHERE YOU ARE

pwd
prints working directory. Starting from the root (/) directory, pwd lists the genealogy of the current directory, ending with the current directory's name. This whole construct is called the pathname. When in doubt, specify a file or directory by using its entire pathname.

EDIT TEXT OR PROGRAMS

ex or vi <filename>.
If you are intimidated by all of the ex options, use its subset, edit. vi is the screen-oriented version of ex.

FORMAT TEXT

nroff -ms <filenames>
The nroff program with the "ms" macro commands is the easiest way to format text neatly and uniformly. Other macro packages are available, and straight nroff can be used for "special effects". You can also define your own macro formatting commands.

VIEW OUTPUT ON SCREEN	more <filename> Alternatively, use the commands 'cat' or 'nroff' and pipe the output through the <u>more</u> program, as in: nroff filename | more This will put the output on your CRT one screenful at a time. Hit the space bar to get the next screenful, and Shift/Delete, to exit.
STRING COMMANDS TOGETHER	You can pipe the output of one command to the input of another with the pipe "|" sign, as for the "more" program above. Commands can also be performed sequentially if they are separated by semicolons ";". It is usually best to confine a string of commands to one line on the screen or printer. Finish all commands with a carriage return.
EXIT	To stop a program and exit to your shell (prompt) press the "Delete" key. To log out, type Control-D. To stop a running program abruptly, type Control-|. This "quit" signal creates a core image of the program that you interrupted, which may be used for diagnosis.

Common Errors and How to Fix Them

1. Ls or other terminal output is bunched up (seems to be missing tabs)

Cure: Type tset.

2. The terminal is not echoing or seems to be dead.

Cure: Type "Linefeed" - Control-j on terminals without a linefeed rather than "Return". If you get a prompt, type tset and Linefeed.

3. Programs that are likely to access raw devices, such as read, write, and lseek, should always be given parameters in 512-byte multiples, since in raw I/O read and write truncate file offsets to 512-byte block boundaries. Write, in particular, scribbles on the tail of incomplete blocks.

User Documentation Update for UNISOFT Pascal and FORTRAN

1. The close procedure from Pascal is always "lock" (the file remains after the close) regardless of whether "lock" or "purge" is specified. Similarly, from FORTRAN, all files are closed "keep" even if the "delete" option is specified.

2. The following calls are not implemented under the UNISOFT version of SVS Pascal: unitread, unitwrite, unitclear, unitstatus, and memavail.

3. Pascal programs must be in files whose names end in ".pas" FORTRAN programs must be in files whose names end in ".for".

4. Call "C" externals like the following example:

Provide an external definition in Pascal program: (assume the pchar is declared ^char)

```
function _write(count: longint;
                bufaddr: pchar;
                fd: longint): longint; external;
```

Note: arguments are in reverse order from "C" call and all arguments must be declared in Pascal to push 4 bytes onto the stack for the call.

A "wrapper" must be provided in assembler language. The external reference passed to the UNISOFT linker will be in upper case (_WRITE). The wrapper must call the corresponding lower case routine and get the return value out of D0 and onto the stack where Pascal expects it. An example of a proper wrapper for _write is as follows:

```
        .globl _WRITE
_WRITE: movl   sp@+,a3    | Save return address
        jsr    _write     | Call "C" style routine
        addl   #12,sp     | Remove 12 bytes of arguments
        movl   d0,sp@     | Place return value on stack
        jmp    a3@        | Return to caller
```

Assemble the wrapper into a .o file using the UNISOFT assembler and provide it to the UNISOFT linker (cc in the sample shell command file) next to wraplib.o.

5. Calling "C" externals from FORTRAN is accomplished by simply calling them as a function. A wrapper (as above) must be provided Parameters are passed by FORTRAN by reference so the wrapper (or called routine) should expect pointers to the arguments to be passed. For example, calling ICFUNCT:

```
      INTEGER ICFUNCT,I,M,N,O
      I = ICFUNCT(M,N,O)
```

will generate an external reference for the UNISOFT linker if it i not resolved by ulinker to another Pascal or FORTRAN routine.

6. A Pascal program may call halt(0) to generate an UNISOFT error return and halt(1) to generate a normal UNISOFT termination if the program is used in scripts which test the UNISOFT error flag.

INTRODUCTION TO VOLUME 1

This volume describes the user-accessible facilities of the UNIX* operating system.

Volume One is the User's Manual. This volume includes short descriptions of commands, subroutines, system calls, and other useful information.

Volumes Two and Three contain tutorials and reference articles for other UNIX functions such as systems programming and document preparation.

Volume Three, in particular, contains a good introductory document, "The UNIX Time-Sharing System" by Dennis Ritchie and Ken Thompson. A beginners' UNIX tutorial is found in "UNIX for Beginners" by Brian Kernighan.

Within the area it surveys, this User's Manual (Volume One) attempts to be timely, complete and concise. The supplementary documents in the other volumes will often clarify fine points of syntax or usage that the short descriptions omit, for the sake of brevity. However, the short descriptions with their examples should be sufficient to show the common usage of most commands and other facilities. It is intended that each program be described as it is, not as it should be.

Volume One is divided into eight sections:

1. Commands
2. System calls
3. Subroutines
4. Special files
5. File formats and conventions
6. Games
7. Macro packages and language conventions
8. Maintenance commands and procedures

Commands are programs intended to be invoked directly by the user, in contrast 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 interpreters, *sh* and *csh*. Each user has the option of using either the Bourne shell, (*sh*) or the C-Shell (*csh*) as the usual command interpreter.

System calls are entries into the UNIX supervisor. The system call interface is identical to a C language program

*UNIX is a Trademark of Bell Laboratories.

call; notes on system calls are found in Section 2.

An assortment of subroutines is available; they are described in section 3. The primary libraries in which they are kept are described in intro(3). Subroutines, like system calls, are described in terms of the C programming language.

The special files section 4 discusses the characteristics of system "files" which are symbolic representations of physical I/O devices, such as terminals (see tty(4)).

Section 5, concerning file formats and conventions, details the structure and characteristics of system files used for diagnostics or as automatic holding files for the output of the loader or the assembler.

Games have been relegated to section 6 to keep them from contaminating the more staid information of section 1.

Section 7 is a miscellaneous collection of information necessary to writing in various specialized languages: character codes, macro packages for typesetting, etc.

Section 8, on maintenance, discusses commands and procedures used for system maintenance and/or diagnosis. These maintenance features are usually used in "super-user" mode or by a system administrator. Maintenance commands and files are almost all kept in the directory /etc.

Each of the sections of Volume One, consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, together with the section number. Entries within each section are alphabetized. The page numbers of each entry start at 1; to aid in adding updates or revision, each entry has been numbered separately.

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

The name subsection lists the exact names of the commands and subroutines covered under the entry and gives a very short description of their purpose.

The synopsis summarizes the use of the program being described. A few conventions are used, particularly in the Commands section, Section 1:

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 option-specifying argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with ´-´.

The _description_ subsection discusses in detail the subject at hand.

The _example_ subsection gives one or more sample uses of the command or program.

The _files_ subsection gives the names of files which are built into the program.

A _see also_ subsection gives pointers to related information.

A _diagnostics_ subsection discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.

The _bugs_ subsection gives known bugs and sometimes deficiencies. Occasionally also the suggested fix is described.

At the beginning of the volume 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.

HOW TO GET STARTED

This section sketches 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" in Volume 2 for a more complete introduction to the system.

<u>Logging in</u>.

After the system has booted up and you are running the shell program with a login : prompt, type your login name. 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, namely a carriage return + a line feed. Always type your login name in lower-case if possible. If you type it in in upper-case letters, UNIX will assume that your terminal cannot generate lower-case letters and will translate all subsequent lower-case letters to upper case.

The evidence that you have successfully logged in is that a shell program will type the C-shell prompt ('%') to you. The shells are described below under "How to run a Program" and in <u>csh</u>(<u>1</u>) and <u>sh</u>(<u>1</u>) in Section 1.

For information on setting up terminals, consult <u>tset</u>(<u>1</u>), and <u>stty</u>(<u>1</u>), which tell how to adjust terminal behavior. <u>Getty</u>(<u>8</u>) discusses the login sequence in more detail, and <u>tty</u>(<u>4</u>), discusses terminal I/O.

<u>Logging out</u>.

There are two ways to 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.

Or, another user can log in directly after you by giving a <u>login</u>(<u>1</u>) command.

<u>How to communicate through your terminal</u>.

When you type characters to UNIX, the system stores all the incoming characters in a buffer until a carriage return is hit. The characters will not be given to a program until you type a return (or newline), as described above in <u>Logging in</u>.

UNIX terminal 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 printed 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 (or beeps, if your prompt was a %).

The character "@" in typed input kills all the preceding characters in the line, so typing mistakes can be repaired on a single line. Also, the character "#" erases the last character typed. (Most users prefer to use a backspace rather than "#", and many prefer control-U instead of "@"; _tset_(_1_) or _stty_(_1_) can be used to arrange this.) 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 ´break´ or ´interrupt´ key causes an _interrupt signal_, as does the ASCII ´delete´ (or ´rubout´) character, which is not passed to programs. 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. Many users change this interrupt character to be ^C (control-C) using _stty_(1).

It is also possible to suspend output temporarily using ^S (control-s) and later resume output with ^Q.

The _quit_ or "abort" signal is generated by typing the ASCII FS character. (FS appears many places on different terminals, most commonly as control-\ or control-|.) It not only causes a running program to terminate abruptly, but also generates a file with the core image of the terminated process. _Quit_ is therefore useful for debugging (see also _core_(_5_)).

Besides adapting to the speed of the terminal, UNIX tries to be intelligent about whether you have a terminal with the newline function or whether it must be simulated with carriage-return and line-feed. In the latter case, all input carriage returns are turned to newline 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, _stty_(_1_) or _tset_(_1_) can be used to reset your terminal.

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 _tset_(_1_) or _stty_(_1_) command will set or reset this mode. _Tset_(_1_) can be used to set the tab stops automatically when necessary.

How to Run a Program: the Shells.

When you have successfully logged in, a program called a 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 in several system directories to find the command. You can also place commands in your own directory and have the shell find them there. 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, one space between each separate element.

When a program terminates, the shell will ordinarily regain control and type a prompt at you to indicate that it is ready for another command.

The shells have many other capabilities, which are described in detail in sections _sh_(_1_) and _csh_(_1_). See also the reference articles on the Bourne shell and the C-shell.

The current directory.

UNIX has a file system arranged in a hierarchy of directories. Initially, you have one login directory which has the same name as your login name. When you log in, any file name you type is by default entered in this directory. Since you are the owner of this directory, you have full permission 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 perusal by other users. See also chmod(1).

To change the current directory (but not the set of permissions you were endowed with at login) use _cd_(_1_).

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. For example, /_unisoft_/_lem_/_filex_ refers to the file _filex_ in the directory _lem_; _lem_ is itself a subdirectory of _unisoft_; _unisoft_ springs directly from the root directory, /.

If your current directory has subdirectories, the path names of files therein begin with the name of the subdirectory with no prefixed "/".

A path name may be used anywhere a file name is required.

Important commands which modify the contents of files are _cp_(_1_), _mv_(_1_), and _rm_(_1_), which respectively copy, move (i.e. rename) and remove files. To find out the status of files or directories, use _ls_(_1_). See _mkdir_(_1_) for making directories and _rmdir_ (in _rm_(_1_) for destroying them.

For a fuller discussion of the file system, see "The UNIX Time-Sharing System," by Ken Thompson and Dennis Ritchie. It may also be useful to glance through section 2 of this manual, which discusses system calls, even if you don't intend to deal with the system at that level. The Introduction to Section 2 also contains a list of error messages.

Writing a program.

To enter the text of a source program into a UNIX file, use the editor _ex_(_1_) or its display editing alias _vi_(_1_). (The old standard editor _ed_(_1_) is also available.) The principal languages in UNIX are provided by the C compiler _cc_(_1_), the Fortran compiler, and the Pascal compiler. 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 change the name from _a.out_ to something else, since _a.out_ is subject to being written over at the next compiler call).

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 shell ('%') prompt.

Your programs can receive arguments from the command line just as system programs do: see _exec_(2).

Text processing.

Almost all text is entered through the editor _ex_(_1_) (often entered via _vi_(_1_)). The commands most often used to output text on a terminal or printer are: _cat_, _pr_, _more_ and _nroff_, all in section 1.

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 multicolumn output. _Nroff_ is an elaborate text formatting program. Used naked, it requires careful forethought, but for ordinary documents it can be used through a macro package such as _me_ or _ms_, which are described in section 7.

Troff prepares documents for a Graphics Systems phototypesetter or a Versatec Plotter; it is very similar to _nroff_, and often works from exactly the same source text.

More(_1_) is useful for viewing a long text on a CRT screen one page at a time. It helps prevent the output of a command from zipping off the top of your screen. It is also well suited to perusing files. The output from any set of commands can be piped through _more_ in order to be viewed on a CRT screen; see "Pipes and Filters" in _csh_(_1_).

Status inquiries.

Various commands exist to provide you with useful information. For example, _date_(_1_) prints the current time and date. _ls_(_1_) will list the files in your directory or give summary information about particular files.

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_(_1_) is used; _mail_(_1_) 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.

NAME

intro - introduction to commands

DESCRIPTION

Section 1 of the Programmers Manual contains short descriptions and examples of commands used directly at the user interface level. The commands appear in alphabetic order.

SEE ALSO

Section (6) for computer games.

How to get started, in the Introduction.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of 'normal' termination) one supplied by the program, see *wait*(1) and *exit*(2). The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously "exit code", "exit status" or "return code", and is described only where special conventions are involved.

NAME

adb - debugger

SYNOPSIS

adb [-w] [objfil [corfil]]

DESCRIPTION

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

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

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

To EXIT adb: use $q or $Q or Control-d.

In general requests to *adb* are of the form

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

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

The interpretation of an address depends on the context it is used in. If a subprocess is being debugged then addresses are interpreted in the usual way in the address space of the subprocess. If the operating system is being debugged either post-mortem or using the special file /dev/kmem to interactive examine and/or modify memory the maps are set to map the kernel virtual addresses. For further details of address mapping see ADDRESSES.

EXPRESSIONS

. The value of *dot*.

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

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

" The last *address* typed.

integer

A number. The prefix 0 (zero) forces interpretation in octal radix; the prefixes 0d and 0D force interpretation in decimal radix; the prefixes 0x and 0X force interpretation in hexadecimal radix. Thus 020 = 0d16 = 0x10 = sixteen. If no prefix appears, then the *default radix* is used; see the $d command. The default radix is initially hexadecimal. The hexadecimal digits are 0123456789abcdefABCDEF with the obvious values. Note that a hexadecimal number whose most significant digit would otherwise be an alphabetic character must have a 0x (or 0X) prefix (or a leading zero if the default radix is hexadecimal).

integer.*fraction*
A 32 bit floating point number.

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

< *name* The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (see VARIABLES) named by single letters or digits. If *name* is a register name then the value of the register is obtained from the system header in *corfil*. The register names are those printed by the $r command.

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

_ *symbol*
In C, the 'true name' of an external symbol begins with _. It may be necessary to utter this name to distinguish it from internal or hidden variables of a program.

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

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

Monadic operators

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

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

-*exp* Integer negation.

~*exp* Bitwise complement.

#*exp* Logical negation.

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

_e1_+_e2_ Integer addition.

_e1_-_e2_ Integer subtraction.

_e1_*_e2_ Integer multiplication.

_e1_%_e2_ Integer division.

_e1_&_e2_ Bitwise conjunction.

_e1_|_e2_ Bitwise disjunction.

_e1_#_e2_ _E1_ rounded up to the next multiple of _e2_.

COMMANDS

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

?_f_ Locations starting at _address_ in _objfil_ are printed according to the format _f_. _dot_ is incremented by the sum of the increments for each format letter (q.v.).

/_f_ Locations starting at _address_ in _corfil_ are printed according to the format _f_ and _dot_ is incremented as for '?'.

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

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

i	n	Disassemble the addressed instruction.
o	2	Print 2 bytes in octal. All octal numbers output by _adb_ are preceded by 0.
O	4	Print 4 bytes in octal.
q	2	Print in signed octal.
Q	4	Print long signed octal.
d	2	Print in decimal.
D	4	Print long decimal.
x	2	Print 2 bytes in hexadecimal.
X	4	Print 4 bytes in hexadecimal.
u	2	Print as an unsigned decimal number.
U	4	Print long unsigned decimal.

f 4 Print the 32 bit value as a floating point number.
F 8 Print double floating point.
b 1 Print the addressed byte in octal.
c 1 Print the addressed character.
C 1 Print the addressed character using the standard escape convention where control characters are printed as ^X and the delete character is printed as ^?.
s _n_ Print the addressed characters until a zero character is reached.
S _n_ Print a string using the ^_X_ escape convention (see C above). _n_ is the length of the string including its zero terminator.
Y 4 Print 4 bytes in date format (see _ctime_(3)).
a 0 Print the value of _dot_ in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

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

p 4 Print the addressed value in symbolic form using the same rules for symbol lookup as a.
t 0 When preceded by an integer tabs to the next appropriate tab stop. For example, 8t moves to the next 8-space tab stop.
r 0 Print a space.
n 0 Print a newline.
"..." 0
Print the enclosed string.
^ _Dot_ is decremented by the current increment. Nothing is printed.
\+ _Dot_ is incremented by 1. Nothing is printed.
\- _Dot_ is decremented by 1. Nothing is printed.

newline
Repeat the previous command with a _count_ of 1.

[?/]l _value_ _mask_
Words starting at _dot_ are masked with _mask_ and compared with _value_ until a match is found. If L is used then the match is for 4 bytes at a time instead of 2. If no match is found then _dot_ is unchanged; otherwise _dot_ is set to the matched location. If _mask_ is omitted then -1 is used.

[?/]w _value_ ...
Write the 2-byte _value_ into the addressed location. If the command is W, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]m _b1_ _e1_ _f1_[?/]
New values for (_b1_, _e1_, _f1_) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second

segment (_b2_,_e2_,_f2_) of the mapping is changed. If the list is terminated by '?' or '/' then the file (_objfil_ or _corfil_ respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to _objfil_.)

>_name_ _Dot_ is assigned to the variable or register named.

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

$_modifier_
Miscellaneous commands. The available _modifiers_ are:

- <_f_ Read commands from the file _f_. If this command is executed in a file, further commands in the file are not seen. If _f_ is omitted, the current input stream is terminated. If a _count_ is given, and is zero, the command will be ignored. The value of the count will be placed in variable _9_ before the first command in _f_ is executed.
- <<_f_ Similar to < except it can be used in a file of commands without causing the file to be closed. Variable _9_ is saved during the execution of this command, and restored when it completes. There is a (small) finite limit to the number of << files that can be open at once.
- >_f_ Append output to the file _f_, which is created if it does not exist. If _f_ is omitted, output is returned to the terminal.
- ? Print process id, the signal which caused stoppage or termination, as well as the registers as $r. This is the default if _modifier_ is omitted.
- r Print the general registers and the instruction addressed by pc. _Dot_ is set to pc.
- b Print all breakpoints and their associated counts and commands.
- c C stack backtrace. If _address_ is given then it is taken as the address of the current frame (instead of a7). If C is used then the names and (16 bit) values of all automatic and static variables are printed for each active function. If _count_ is given then only the first _count_ frames are printed.
- d Set the default radix to _address_ and report the new value. Note that _address_ is interpreted in the (old) current radix. Thus 10$d never changes the default radix. To make decimal the default radix, use 0t10$d.
- e The names and values of external variables are printed.
- w Set the page width for output to _address_ (default 80).
- s Set the limit for symbol matches to _address_ (default 255).
- o All integers input are regarded as octal.
- d Reset integer input as described in EXPRESSIONS.
- q Exit from _adb_.
- v Print all non zero variables in octal.
- m Print the address map.

:_modifier_
Manage a subprocess. Available modifiers are:

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

d Delete breakpoint at *address*.

r Run *objfil* as a subprocess. If *address* is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

c*s* The subprocess is continued with signal *s* c *s*, see *signal*(2). If *address* is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for r.

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

k The current subprocess, if any, is terminated.

VARIABLES

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

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.
9 The count on the last $< or $<< command.

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

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

ADDRESSES

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

b1<*address*<*e1* => *file address*=*address*+*f1*-*b1*, otherwise,

b2<*address*<*e2* => *file address*=*address*+*f2*-*b2*,

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

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

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

FILES

a.out
core

SEE ALSO

a.out(5), core(5)

DIAGNOSTICS

"Adb" when there is no current command or format. Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero status.

BUGS

Use of # for the unary logical negation operator is peculiar.

There doesn't seem to be any way to clear all breakpoints.

NAME

admin - create and administer SCCS files

SYNOPSIS

admin [-n] [-i[name]] [-rrel] [-t[name]]
[-fflag[flag-val]] [-dflag[flag-val]]
[-alogin] [-elogin] [-m[mrlist]] [-y[comment]] [-h] [-z] files

DESCRIPTION

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

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

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

-n This keyletter indicates that a new SCCS file is to be created.

-i[_name_] The _name_ of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see -r keyletter for delta numbering scheme). If the i keyletter is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this keyletter is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an _admin_ command on which the i keyletter is supplied. Using a single _admin_ to create two or more SCCS files require that they be created empty (no -i keyletter). Note that the -i keyletter implies the -n keyletter.

-r_rel_ The _rel_ease into which the initial delta is inserted. This keyletter may be used only if the -i keyletter is also used. If the -r keyletter is not used, the initial delta is inserted into release 1.

The level of the initial delta is always 1 (by default initial deltas are named 1.1).

-t[_name_] The _name_ of a file from which descriptive text for the SCCS file is to be taken. If the -t keyletter is used and _admin_ is creating a new SCCS file (the -n and/or -i keyletters also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a -t keyletter without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a -t keyletter with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

-f_flag_ This keyletter specifies a _flag_, and, possibly, a value for the _flag_, to be placed in the SCCS file. Several f keyletters may be supplied on a single _admin_ command line. The allowable _flag_s and their values are:

b Allows use of the -b keyletter on a _get_(1) command to create branch deltas.

c_ceil_ The highest release (that is, "ceiling"), a number less than or equal to 9999, which may be retrieved by a _get_(1) command for editing. The default value for an unspecified c flag is 9999.

f_floor_ The lowest release (that is, "floor"), a number greater than 0 but less than 9999, which may be retrieved by a _get_(1) command for editing. The default value for an unspecified f flag is 1.

d_SID_ The default delta number (SID) to be used by a _get_(1) command.

i Causes the "No id keywords (ge6)" message issued by _get_(1) or _delta_(1) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see _get_(1)) are found in the text retrieved or stored in the SCCS file.

j Allows concurrent _get_(1) commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l_list_ A _list_ of releases to which deltas can no longer be made (get -e against one of these "locked" releases fails). The _list_ has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= RELEASE NUMBER | a

The character a in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n — Causes *delta*(1) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file preventing branch deltas from being created from them in the future.

q*text* — User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by *get*(1).

m*mod* — *Mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by *get*(1). If the m flag is not specified, the value assigned is the name of the SCCS file with the leading s. removed.

t*type* — *Type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by *get*(1).

v[*pgm*] — Causes *delta*(1) to prompt for Modification Request (*MR*) numbers as the reason for creating a delta. The optional value specifies the name of an *MR* number validity checking program (see *delta*(1)). (If this flag is set when creating an SCCS file, the m keyletter must also be used even if its value is null).

-d*flag* — Causes removal (deletion) of the specified *flag* from an SCCS file. The -d keyletter may be specified only when processing existing SCCS files. Several -d keyletters may be supplied on a single *admin* command. See the -f keyletter for allowable *flag* names.

l*list* — A *list* of releases to be "unlocked". See the -f keyletter for a description of the l flag and the syntax of a *list*.

-a*login* — A *login* name, or numerical UNIX group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all _login_ names common to that group ID. Several a keyletters may be used on a single _admin_ command line. As many _logins_, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas.

-e_login_ A _login_ name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all _login_ names common to that group ID. Several e keyletters may be used on a single _admin_ command line.

-y[_comment_] The _comment_ text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of _delta_(1). Omission of the -y keyletter results in a default comment line being inserted in the form:
date and time created _YY_/_MM_/_DD_ _HH_:_MM_:_SS_ by _login_
The -y keyletter is valid only if the -i and/or -n keyletters are specified (that is, a new SCCS file is being created).

-m[_mrlist_] The list of Modification Requests (_MR_) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to _delta_(1). The v flag must be set and the _MR_ numbers are validated if the v flag has a value (the name of an _MR_ number validation program). Diagnostics will occur if the v flag is not set or _MR_ validation fails.

-h Causes _admin_ to check the structure of the SCCS file (see _sccsfile_(5)), and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This keyletter inhibits writing on the file, so that it nullifies the effect of any other keyletters supplied, and is, therefore, only meaningful when processing existing files.

-z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see -h, above).

Note that use of this keyletter on a truly corrupted file may prevent future detection of the corruption.

FILES

The last component of all SCCS file names must be of the form s.*file-name*. New SCCS files are given mode 444 (see *chmod*(1)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called x.*file-name*, (see *get*(1)), created with mode 444 if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

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

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of *ed*(1). *Care must be taken*! The edited file should *always* be processed by an admin -h to check for corruption followed by an admin -z to generate a proper check-sum. Another admin -h is recommended to ensure the SCCS file is valid.

Admin also makes use of a transient lock file (called z.*file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(1) for further information.

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ar - archive and library maintainer

SYNOPSIS

ar [uvbail] [mrxtdpq] [posname] archivename filename(s) ...

DESCRIPTION

The archive command _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. However, _ar_ can be used for any similar archiving purpose. Archives often consist of unlinked program modules.

Key is _one_ character from the set mrxtdpq, optionally concatenated with one or more of uvnbail. _Archivename_ is the archive file. The _filename(s)_ are constituent files in or destined for the archive file. The meanings of the _key_ characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character u is used with r, then only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set abi is used, then the _posname_ argument must be present and specifies that new files are to be placed after (a) or before (b or i) _posname_. Otherwise new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

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

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the _posname_ argument must be present and, as in r, specifies where the files are to be moved.

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

v Verbose. Under the verbose option, _ar_ gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with t, it gives a long listing of all information about the files. When used with p, it precedes each file with a name.

c Create. Normally *ar* will create *afile* when it needs to. The create option suppresses the normal message that is produced when *afile* is created.

l Local. Normally *ar* places its temporary files in the directory /tmp. This option causes them to be placed in the local directory.

EXAMPLE

```
ar rv libar.a text.o
```

places file text.o in archive libar.a.

```
ar bm file1 archivename file2
```

changes the location of a file inside an archive. File2 is the file to be moved. File2 is moved to a new position before file1.

FILES

/tmp temporaries

SEE ALSO

ld(1), ar(5)

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.
Sufficient disk space must be present to make an entire copy of the archive or the *ar* command will fail.

NAME

as - assembler

SYNOPSIS

as [-o objfile] [-l] [name ...]

DESCRIPTION

_As_ assembles the named files, or the standard input if no file name is specified.

All undefined symbols in the assembly are treated as global.

The relocatable output of the assembly is left on the file _objfile_; if that is omitted, _a.out_ is used.

The -_l_ option produces an assembly listing on file objfile.lst. If the -_l_ option is specified and no -_o_ parameter is specified, the assembly listing is placed on _a.lst_.

EXAMPLE

```
as -o file.o filea fileb filec
```

would assemble the three named files and put the output of the assembly into _file.o_.

FILES

/tmp/as*	default temporary file
a.out	default resultant object file
a.lst	default assembly listing file

SEE ALSO

ld(1), nm(1), adb(1), a.out(5)

NAME

asm - motorola format assembler

SYNOPSIS

asm [-o objfile] [-l] [name ...]

DESCRIPTION

_As_ assembles the named files.

All undefined symbols in the assembly are treated as global.

The relocatable output of the assembly is left on the file _objfile_; if that is omitted, _a.out_ is used.

The -_l_ option produces an assembly listing on file _a.lst_.

EXAMPLE

```
as -o file.o filea fileb filec
```

would assemble the three named files and put the output of the assembly into _file.o_.

FILES

.tmp*	default temporary file
a.out	default resultant object file
a.lst	default assembly listing file

SEE ALSO

ld(1), nm(1), adb(1), a.out(5)

NAME
asmcvt - assembler format converter (MIT to Motorola)

SYNOPSIS
asmcvt fromfile tofile

DESCRIPTION
Asmcvt copys *fromfile* to *tofile* converting anything it belives to be in MIT 68000 assembler format to Motorola assembler format.

The file *tofile* will be overwritten if it exists.

EXAMPLE
```
asmcvt file.s file.m
```

would convert file.s to Motorola format and leave the result in file.m.

SEE ALSO
as(1), asm(1)

BUGS
Not all constructs are recognized, but most of the compiler output should convert with no trouble.

The location counter symbol '.' is not converted.

NAME

at - execute commands at a later time

SYNOPSIS

at time [day] [file]

DESCRIPTION

At squirrels away a copy of the named *file* (standard input default) to be used as input to *sh*(1) at a specified later time. A *cd*(1) command to the current directory is inserted at the beginning, followed by assignments to all environment variables. When the script is run, it uses the user and group ID of the creator of the copy file.

The *time* is 1 to 4 digits, with an optional following "A", "P", "N" or "M" for AM, PM, noon or midnight. One and two digit numbers are taken to be hours, three and four digits to be hours and minutes. If no letters follow the digits, a 24 hour clock time is understood.

The optional *day* is either (1) a month name followed by a day number, or (2) a day of the week; if the word "week" follows invocation is moved seven days further off. Names of months and days may be recognizably truncated. Examples of legitimate commands are

```
at 8am jan 24
at 1530 fr week
```

At programs are executed by periodic execution of the command /*usr*/*lib*/*atrun* from *cron*(1M). The granularity of *at* depends upon how often *atrun* is executed.

Standard output or error output is lost unless redirected.

FILES

/usr/spool/at/yy.ddd.hhhh.uu	activity to be performed at hour *hhhh* of day *ddd* of year *yy*. *uu* is a unique number.
/usr/spool/at/lasttimedone	contains *hhhh* for last hour of activity.
/usr/spool/at/past	directory of activities now in progress.
/usr/lib/atrun	program that executes activities that are due.
/usr/lib/crontab	cron table entry for running atrun.

SEE ALSO

calendar(1), cron(1M)

DIAGNOSTICS

Complains about various syntax errors and times out of range.

BUGS

Due to the granularity of the execution of /*usr*/*lib*/*atrun*, there may be bugs in scheduling things almost exactly 24 hours into the future.

NAME

awk - pattern scanning and processing language

SYNOPSIS

awk [-F*c*] [pattern { action }] [file] ...

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in the *pattern* { *action* } program. With each pattern in *pattern* { *action* } there can be an associated action that will be performed when a line of a *file* matches the pattern. If no action is specified, the lines that qualify will be printed on the standard output.

Patterns may be specified on the command line, or they may be taken from an *awk* command file used with the -f *file* option.

Files to be examined are read in order; if there are no files named, the standard input is read. The option '-' means to use the standard input.

Each line from the files is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, *vide infra*.) The fields are denoted $1, $2, ... In contrast to some other programs in which "0" is the first field, in *awk* $0 refers to the entire line.

A pattern-action statement has the form

    pattern { action }

The "pattern" should be enclosed in double quotation marks if it is a string, and 0 should also be added to the "pattern" to force it to be explicitly treated as a number.

A missing { action } means print the line; a missing pattern always matches.

Patterns may be arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions.

Regular expressions must be surrounded by slashes, and the syntax and metacharacters (as well as the need to escape the metacharacters) follows the same general syntax as does *egrep*.

If the shell complains, also enclose the expressions in double quotation marks.

Isolated regular expressions in a pattern apply to the entire line.

A pattern may also consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second. The action is performed recursively for all such /start/, /stop/ pairs in the file.

Regular expressions may also be used in relational expressions.

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

where a relop is any of the six relational operators in C, and a matchop is either ~ (for contains) or !~ (for does not contain). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character _c_ may be used to separate the fields by starting the program with

```
BEGIN { FS = "c" }
```

or by using the -F_c_ option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default "%.6g").

An action is a sequence of statements. The statements should be connected with a backslash before each newline, if they occupy more than one command line.

A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next  # skip remaining patterns on this input line
exit  # skip the rest of the input
```

Action statements are terminated by semicolons, newlines or right braces. Be sure to escape the newline with a backslash immediately preceding it. Beginning and ending curly braces should be escaped with single quotation marks, one before the opening brace and one immediately after the closing brace. (see EXAMPLES, below). That is, enclose the entire action statement in single quotation marks '{ action }' in order not to be trapped by the shell.

An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, -, *, /, %, and concatenation (indicated by a blank). The C operators ++, --, +=, -=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants must be quoted "...".

The _print_ statement prints its arguments on the standard output (or on a file if >_file_ is present), separated by the current output field separator, and terminated by the output record separator. The _printf_ statement formats its expression list according to the format (see _printf_(3)).

The built-in function _length_ returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions _exp_, _log_, _sqrt_, and _int_. The last truncates its argument to an integer. _substr_(_s_, _m_, _n_) returns the _n_-character substring of _s_ that begins at position _m_. The function _sprintf_ (_fmt_, _expr_, _expr_, ...) (Reg.)formats the expressions according to the _printf_(3) format given by _fmt_ and returns the resulting string.

EXAMPLES

```
awk "length > 72" filea
```

would print lines longer than 72 characters on the standard output.

```
awk '{ print $2, $1 }' filea
```

would print the first two fields of each line in opposite order.

```
awk '{ s += $1 } END {print "sum is", s, "average is", s/NR }' filea
```

would add up the first column and print the sum and average.

```
awk '{ for (i = NF; i > 0; --i) print $1 }' filea
```

would print all the fields of each line in reverse order. The output prints one field per line, beginning at the end of the file, unless otherwise directed.

```
awk "/start/, /stop/" filea
```

would print all lines between start/stop pattern pairs, for every such pair in the file.

FILES

/usr/lib/awklist error log for awk scripts

SEE ALSO

egrep(1), lex(1), sed(1)
A. V. Aho, B. W. Kernighan, P. J. Weinberger, _Awk - a pattern scanning and processing language_

BUGS

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate "" to it.

NAME

badblk - program to set or update bad block information

SYNOPSIS

badblk [-w] [-m N] /dev/rXYZ [#S]

DESCRIPTION

Badblk sets or updates bad block information.

If invoked with the -*w* option, write/verify is performed to determine if there is a bad block; otherwise only read is done.

If invoked with the -*m*N option, the number of alternate blocks will be set to N. *Badblk* panics if N > NICALT (currently 70).

/dev/rXYZ is the device name.

#S is one or more block numbers separated by blanks.

If invoked with no specific block numbers and no bad block verification has been done before, then each block on the disk is checked (either read or write/verify) and bad block information in block 0 is set up from scratch.

If invoked with no specific block numbers, but block 0 already contains bad block information set up earlier, then a verification on the whole disk is performed; any new bad blocks not already on the block 0 table will be added.

If invoked with the device name plus block numbers, then only the indicated blocks are updated in block 0.

After alternate blocks are assigned, block 0 is updated and the updated blocks are verified to make sure alternate blocks are good. If alternate blocks are not good, new alternate block numbers are assigned.

The raw device that accesses the entire disk and allows for writing block zero should be specified.

EXAMPLE

badblk -w /dev/rw1hw0

do a full write/verify on winchester 1 and update the header block. The rw1hw0 specifies raw (r) winchester 1 (w1), the full disk (h), with the capability of writing block 0 (w0).

badblk /dev/rw1hw0 3754 8123

add blocks 3754 and 8123 to the badblock list.

NAME

basename - strip filename affixes

SYNOPSIS

basename string [suffix]

DESCRIPTION

Basename deletes any prefix ending in `/' and the *suffix*, if present in *string*, from *string*, and prints the result on the standard output. It is normally used inside substitution marks ` ` in shell procedures.

EXAMPLE

This shell procedure invoked with the argument /*usr*/*src*/*cmd*/*cat*.*c* compiles the named file and moves the output to *cat* in the current directory:

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

SEE ALSO

sh(1)

NAME

bc - arbitrary-precision arithmetic language

SYNOPSIS

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

DESCRIPTION

_Bc_ is an interactive processor for a language that resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The -l argument stands for the name of an arbitrary precision math library. The syntax for _bc_ programs is as follows; L means letter a-z, E means expression, S means statement.

Comments
```
are enclosed in /* and */.
```

Names
```
simple variables: L
array elements: L [ E ]
The words ``ibase'', ``obase'', and ``scale''
```

Other operands
```
arbitrarily long numbers with optional sign and decimal point.
( E )
sqrt ( E )
length ( E )      number of significant decimal digits
scale ( E ) number of digits right of decimal point
L ( E , ... , E )
```

Operators
```
+ - * / % ^ (% is remainder; ^ is power)
++  --      (prefix and postfix; apply to names)
== <= >= != < >
= =+ =- =* =/ =% =^
```

Statements
```
E
{ S ; ... ; S }
if ( E ) S
while ( E ) S
for ( E ; E ; E ) S
null statement
break
quit
```

Function definitions
```
define L ( L ,..., L ) {
      auto L, ... , L
      S; ... S
      return ( E )
}
```

Functions in -l math library

```
s(x)   sine
c(x)   cosine
e(x)   exponential
l(x)   log
a(x)   arctangent
j(n,x) Bessel function
```

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or new-lines may separate statements. Assignment to _scale_ influences the number of digits to be retained on arithmetic operations in the manner of _dc_(1). Assignments to _ibase_ or _obase_ set the input and output number radix respectively.

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

_Bc_ is actually a preprocessor for _dc_(1), which it invokes automatically, unless the -c (compile only) option is present. In this case the _dc_ input is sent to the standard output instead.

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
/usr/bin/dc desk calculator proper

SEE ALSO

dc(1).
BC - An Arbitrary Precision Desk-Calculator Language
by L. L. Cherry and R. Morris.

BUGS

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

NAME
bdiff - big diff

SYNOPSIS
bdiff file1 file2 [n] [-s]

DESCRIPTION
Bdiff is used in a manner analogous to *diff*(1) to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for *diff*. *Bdiff* ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. The value of *n* is 3500 by default. If the optional third argument is given, and it is numeric, it is used as the alue for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail. If *file1* (*file2*) is -, the standard input is read. The optional -s (silent) argument specifies that no diagnostics are to be printed by *bdiff* (note, however, that this does not suppress possible exclamations by *diff*. If both optional arguments are specified, they must appear in the order indicated above.

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

FILES
/tmp/bd?????

SEE ALSO
diff(1).

DIAGNOSTICS
Use *help*(1) for explanations.

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.

EXAMPLE

```
cal 9 1752
```

produces a calendar for September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.
Beware that 'cal 82' refers to the early Christian era, not the 20th century.

NAME

calendar - reminder service

SYNOPSIS

calendar [-]

DESCRIPTION

Calendar consults the file calendar in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Most reasonable month-day dates such as \`\`Dec. 7,'' \`\`december 7,'' \`\`12/7,'' etc., are recognized, but not \`\`7 December' or \`\`7/12''. On weekends \`\`tomorrow'' extends through Monday.

When an argument is present, _calendar_ does its job for every user who has a file calendar in his login directory and sends him any positive results by _mail_(1). Normally this is done daily in the wee hours under control of _cron_(1M).

FILES

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

SEE ALSO

cron(1M), mail(1).

BUGS

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

NAME

cat - catenate and print

SYNOPSIS

cat [-u] [-n] [-s] [-v] [-e] [-t] 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. Output is buffered in 512-byte blocks unless the standard output is a terminal, in which case it is line buffered. The -u option causes the output to be completely unbuffered, i.e.: one character at a time.

The option -n causes the output lines to be numbered sequentially from 1. Giving -b with -n causes numbers to be omitted from blank lines.

The option -s causes the output to be single spaced by crushing out multiple adjacent empty lines.

The option -v causes non-printing characters to be printed in a visible way. Control characters print like ^X for control-x; the delete character (octal 0177) prints as ^?. Non-ascii characters (with the high bit set) are printed as M- (for meta) followed by the character of the low 7 bits. A -e option may be given with -v and causes the ends of lines to be followed by the character '$'; the -t option with -v causes tabs to be printed as ^I.

EXAMPLE

```
cat -n filea fileb >> filec
```

numbers the lines of filea and fileb and puts the output in filec.

SEE ALSO

cp(1), ex(1), more(1), pr(1), tail(1)

BUGS

Beware of 'cat a b >a' and 'cat a b >b', which destroy the input files before reading them.

NAME

cb - C program beautifier

SYNOPSIS

cb [file]

DESCRIPTION

Cb places a copy of the C program from the named file, or standard input if no file name is specified, to the standard output with spacing and indentation that displays the structure of the program.

EXAMPLE

If there is a C program called *test.c* which looks like this:

```
#define COMING 1
#define GOING 0


main ()
{
/*  This is a test of the C Beautifier  */
if (COMING)
printf ("Hello, world\n");
else
printf ("Goodbye, world\n");
}
```

Then using the *cb* command as shown below produces the output shown:

```
cb test.c
#define COMING 1
#define GOING 0


main ()
{
        /*  This is a test of the C Beautifier  */
        if (COMING)
                printf ("Hello, world\n");
        else
                printf ("Goodbye, world\n");
}
```

BUGS

Beware of `cb test.c >test.c' which will destroy the input file before reading it.

NAME

cc - C compiler

SYNOPSIS

cc [option] ... file ...

DESCRIPTION

cc is the UNIX C compiler.

cc accepts several 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.

In the same way, arguments whose names end with '.s' are taken to be assembly source programs and are assembled, producing a '.o' file.

The following options are interpreted by cc. See ld(1) for load-time options.

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

-n Passed on to ld to make the text of the resulting program shared.

-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 monitor(3) 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(1).

-O(KPS) Invoke an object-code improver (optimizer). If K is specified, certain UNIX kernel optimizer functions are not performed. If P is specified, stack probe instructions are removed. (NOTE: P should only be used for the operating system source.) If S is specified, stack frame optimization is performed and the debugger, ADB(1), might indicate too few subroutine parameters on stack trace back.

-R (addr) Passed on to ld, making the resulting object module origin'ed at addr(hex).

-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 send the result to the corresponding files suffixed. '.i'

-C prevent the macro preprocessor from eliding (leaving out) comments.

-o *output* Name the final executable output file *output*. If this option is used the file 'a.out' will be left undisturbed.

-D*name*=*def*
-D*name* Define the *name* to the preprocessor, as if by "#define". If no definition is given, the name is defined as "1".

-U*name* Remove any initial definition of *name*.

-I*dir* "#include" files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in -I options, then in the directory /usr/include.

-v print the name of each subprocess as it is executing.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded via LD(1) (in the order given) to produce an executable program with name a.out.

EXAMPLE

```
cc -o output prog1.c prog2.c prog3.c
```

would compile code in the three named C programs and put the compiled code into the file *output*.

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/cpp	preprocessor
/lib/c0	compiler pass1
/lib/c1	compiler pass2
/lib/c2	optional optimizer invoked with "-O"
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	runtime startoff for profiling
/lib/libc.a	standard library, see section 3
/usr/include	standard directory for '#include' files
/lib/libm.a	math library

SEE ALSO

monitor(3), prof(1), adb(1), ld(1), lint(1) B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978

B. W. Kernighan, _Programming in C-a tutorial_
D. M. Ritchie, _C Reference Manual_

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Confusing syntax may cause the "C" compiler to indicate an error on the line following the actual error.

NAME

cd - change working directory

SYNOPSIS

cd directory

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in _directory_. If you are not the owner of a directory and search permission is denied to others, you cannot change to that directory, and the message "Permission denied" will result.

Because a new process is created to execute each command, _cd_ would be ineffective if it were written as a normal command. It is therefore recognized and executed by the shells. In _csh_(1) you may specify a list of directories in which _directory_ is to be sought as a subdirectory if it is not a subdirectory of the current directory; see the description of the _cdpath_ variable in _csh_(1).

EXAMPLE

```
cd /unisoft/usr/games
```

would relocate you to the directory "/unisoft/usr/games" if this directory is executable (searchable) by you.

SEE ALSO

csh(1), sh(1), pwd(1), chdir(2)

NAME

cdc - change the delta commentary of an SCCS delta

SYNOPSIS

cdc -rSID [-m[mrlist]] [-y[comment]] files

DESCRIPTION

Cdc changes the _delta_ _commentary_, for the _SID_ specified by the -r keyletter, of each named SCCS file.

Delta _commentary_ is defined to be the Modification Request (MR) and comment information normally specified via the _delta_(1) command (-m and -y keyletters).

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

Arguments to _cdc_, which may appear in any order, consist of _keyletter_ arguments, and file names.

All the described _keyletter_ arguments apply independently to each named file:

-r_SID_ Used to specify the _S_CCS _ID_entification (_SID_) string of a delta for which the delta commentary is to be changed.

-m[_mrlist_] If the SCCS file has the v flag set (see _admin_(1)) then a list of MR numbers to be added and/or deleted in the delta commentary of the _SID_ specified by the -r keyletter _may_ be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of _delta_(1). In order to delete an MR, precede the MR number with the character ! (see _EXAMPLES_). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If -m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see -y keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the v flag has a value (see *admin*(1)), it is taken to be the name of a program (or shell procedure) which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, *cdc* terminates and the delta commentary remains unchanged.

-y[*comment*] Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the -r keyletter. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

If -y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the *comment* text.

The exact permissions necessary to modify the SCCS file are documented in the *Source Code Control System User's Guide*. Simply stated, they are either (1) if you made the delta, you can change its delta commentary; or (2) if you own the file and directory you can modify the delta commentary.

EXAMPLES

```
cdc -r1.6 -m"b178-12345 !b177-54321 b179-00001" -ytrouble s.file
```

adds b178-12345 and b179-00001 to the MR list, removes b177-54321 from the MR list, and adds the comment trouble to delta 1.6 of s.file.

```
cdc -r1.6 s.file
MRs? !b177-54321 b178-12345 b179-00001
comments? trouble
```

does the same thing.

WARNINGS

If SCCS file names are supplied to the *cdc* command via the standard input (- on the command line), then the -m and -y keyletters must also be used.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use _help_(1) for explanations.

NAME

chgrp - change group

SYNOPSIS

chgrp group file ...

DESCRIPTION

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

Only the super-user can change group.

However, you can often work on a copy of a file by copying it to one of your own directories. See cp(1).

EXAMPLE

```
chgrp unisoft filea fileb filec
```

would put the three files in the "unisoft" group.

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(2), passwd(5), group(5)

NAME
chmod - change mode

SYNOPSIS
chmod mode file ...

DESCRIPTION
The mode of each named file is changed according to _mode_, which may be absolute or symbolic.

An absolute _mode_ is an octal number constructed from the OR-ing (in effect, adding up) of the numbers of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit, see _chmod_(2)
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic _mode_ has the form:

[_who_] _op permission_ [_op permission_] ...

The _who_ part is a combination of the letters u (for user's permissions), g (group) and o (other). The letter a stands for all of the letters "ugo". If _who_ is omitted, the default is _a_ but the setting of the file creation mask is taken into account.

_Op_ can be + to add _permission_ to the file's mode, - to take away _permission_ and = to assign _permission_ absolutely (all other bits will be reset).

Permission is any combination of the letters r (read), w (write), x (execute), s (set owner or group id) and t (save text - sticky). Letters u, g or o indicate that _permission_ is to be taken from the current mode. Omitting _permission_ is only useful with = to take away all permissions.

EXAMPLES
chmod 755 filename

changes the mode of a file you own to: read, write, execute (400+200+100) by owner and read, execute (40+10) for group and read, execute (4+1) for others.

An _ls_ -_l_ of filename shows [-rwxr-xr-x filename] that the requested mode is in effect.

chmod = filename

will take away all permissions from _filename_, including yours.

```
chmod o-w file
```

denies write permission to others.

```
chmod +x file
```

makes a file executable.

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter s is only useful with u or g.

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

SEE ALSO

ls(1), chmod(2), stat(2), umask(2), chown(1M)

NAME

chown - change owner

SYNOPSIS

chown owner file ...

DESCRIPTION

Chown changes the owner of the _files_ to _owner_. The owner may be either a decimal user ID or a login name found in the password file. The password file is /etc/passwd.

Only the super-user can change owner.

However, you can often work on a copy of a file by copying it to one of your own directories. See cp(1).

EXAMPLE

```
chown unisoft filea fileb filec
```

would make "unisoft" the owner of the three files.

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(2), passwd(5), group(5)

NAME

clear - clear terminal screen

SYNOPSIS

clear

DESCRIPTION

Clear clears your screen if this is possible. It looks in the environment for the terminal type and then in /_etc_/_termcap_ to figure out how to clear the screen.

EXAMPLE

```
clear
```

clears the screen.

FILES

/etc/termcap terminal capability data base

NAME

clri - clear i-node

SYNOPSIS

clri filesystem i-number ...

DESCRIPTION

N.B.: *Clri* is made obsolete for normal file system repair work by *fsck*(1M).

Clri writes zeros on the i-nodes with the decimal *i-numbers* on the *filesystem*. After *clri*, any blocks in the affected file will show up as 'missing' in an *fsck*(1) of the *filesystem*.

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

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

SEE ALSO

fsck(1M)

BUGS

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

NAME

col - filter reverse line feeds

SYNOPSIS

col [-bfx]

DESCRIPTION

Col is used for preparing multicolumn output on printers using the *nroff* text formatting package. *Col* enables proper creation of columns by keeping the printer on the same line until all column parts have been printed. It performs the line overlays implied by reverse line feeds (ESC-7 in ASCII) and by forward and reverse half line feeds (ESC-9 and ESC-8). *Col* is particularly useful for filtering multicolumn output made with the '.rt' command of *nroff* and output resulting from use of the *tbl*(1) preprocessor.

Although *col* accepts half line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full line boundary. This treatment can be suppressed by the -f (fine) option; in this case the output from *col* may contain forward half line feeds (ESC-9), but will still never contain either kind of reverse line motion.

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

The control characters SO (ASCII code 017), and SI (016) are assumed to start and end text in an alternate character set. The character set (primary or alternate) associated with each printing character read is remembered; on output, SO and SI characters are generated where necessary to maintain the correct treatment of each character.

Col normally converts white space to tabs to shorten printing time. If the -x option is given, this conversion is suppressed.

All control characters are removed from the input except space, backspace, tab, return, newline, ESC (033) followed by one of 7, 8, 9, SI, SO, and VT (013). This last character is an alternate form of full reverse line feed, for compatibility with some other hardware conventions. All other non-printing characters are ignored.

EXAMPLE

```
nroff -ms filea|col
```

pipes multicolumn *nroff* output through the *col* filter to enable proper creation of columns.

SEE ALSO

troff(1), tbl(1)

BUGS

Col can't back up more than 128 lines. There must not be more than 800 characters, including backspaces, on a line.

NAME

comb - combine SCCS deltas

SYNOPSIS

comb [-o] [-s] [-psid] [-clist] files

DESCRIPTION

Comb generates a shell procedure (see *sh*(1)) which, when run, will reconstruct the given SCCS files. The reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, *comb* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

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

-p*SID* The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-c*list* A *list* (see *get*(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

-o For each get -e generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the -o keyletter may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-s This argument causes *comb* to generate a shell procedure which, when run, will produce a report giving, for each file: the file name, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:
100 * (original - combined) / original
It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

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

FILES

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

SEE ALSO

admin(1), delta(1), get(1), help(1), prs(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use _help_(1) for explanations.

BUGS

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

NAME

comm - select or reject lines common to two sorted files

SYNOPSIS

comm [- [123]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in *file1*; lines only in *file2*; and lines in both files. The filename '-' means the standard input for file 1 (or file 2).

Flags 1, 2, or 3 suppress printing of the corresponding column.

EXAMPLES

```
comm -12 filea fileb
```

prints only the lines common to filea and fileb.

```
comm -23 filea fileb
```

prints only lines in the first file but not in the second.

```
comm -123 filea fileb
```

is not an option, as it suppresses all output.

```
comm -3 filea fileb
```

prints only the lines that differ in the two files.

SEE ALSO

cmp(1), diff(1)

NAME

cp - copy

SYNOPSIS

cp file1 file2

cp file ... directory

DESCRIPTION

File1 is copied onto _file2_. The mode and owner of _file2_ are preserved if it already existed; the mode of the source file is used otherwise.

In the second form, one or more _files_ are copied into the _directory_ with their original file-names.

_Cp_ refuses to copy a file onto itself.

EXAMPLE

```
cp alpha beta gamma /unisoft/barbara
```

places copies of the three files in the directory _barbara_.

SEE ALSO

cat(1), pr(1), mv(1)

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_(1M).

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=monday).

Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by _cron_ every minute.

FILES

/usr/lib/crontab

NAME

crypt - encode/decode

SYNOPSIS

crypt [password]

DESCRIPTION

Crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

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

will print the clear.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; \`sneak paths' by which keys or cleartext can become visible must be minimized.

Crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e. to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lower-case letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. To minimize this possibility, *crypt* takes care to destroy any record of the key immediately upon entry. No doubt the choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty	for typed key
/lib/makekey	to generate a key

SEE ALSO

ed(1), crypt(3), makekey(1)

BUGS

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, neither Bell Telephone Laboratories nor UNISOFT Corporation (Berkeley) assumes any responsibility for their use by the recipient. Further, neither Bell Laboratories nor UNISOFT Corporation (Berkeley) assumes any obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

NAME

csh - a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [-cefinstvVxX] [arg ...]

DESCRIPTION

Csh is a command language interpreter incorporating a history mechanism (see History Substitutions) and a C-like syntax.

An instance of *csh* begins by executing commands from the file '.cshrc' in the *home* directory of the invoker. If this is a login shell then it also executes commands from the file ".login" there. It is typical for users on crt's to put the command "stty crt" in their .*login* file, and to also invoke *tset*(1) there.

In the normal case, the shell will then begin reading commands from the terminal, prompting with '% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates, it executes commands from the file '.logout' in the user's home directory.

LEXICAL STRUCTURE

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters "&" "|" ";" "<" ">" "(" ")" form separate words. If doubled in '&&', '||', '<<' or '>>' these pairs form single words. These parser metacharacters may be made part of other words, or their special meaning may be prevented, by preceding them with a backslash, "\". A newline preceded by a '\' is equivalent to a blank. It is usually necessary to use the backslash to "escape" the parser metacharacters when you want to use them literally rather than as metacharacters.

Strings enclosed in matched pairs of quotation marks, either single or double quotation marks, "'", "`" or """, form parts of a word. Metacharacters in these strings, including blanks and tabs, do not form separate words. Such quotations have semantics to be described subsequently.

Within pairs of single or double quotation marks a newline (carriage return) preceded by a '\' gives a true newline character. This is used to set up a file of strings separated by newlines, as for *fgrep*(*1*).

When the shell's input is not a terminal, the character "#" introduces a comment which continues to the end of the input line. It is prevented from having this special meaning when preceded by '\' or if bracketed by a pair of single or double quotation marks.

COMMANDS

A simple command is a sequence of words, the first of which specifies the command to be executed.

A simple command or a sequence of simple commands separated by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next.

Sequences of pipelines may be separated by ';', and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an "&", which means "run it in background".

Parentheses "(" and ")" around a pipeline or sequence of pipelines cause the whole series to be treated as a simple command, which may in turn be a component of a pipeline, etc. It is also possible to separate pipelines with '||' or '&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See Expressions.)

PROCESS I.D. NUMBERS

When a process is run in background with '&', the shell prints a line which looks like:

    1234

indicating that the process which was started asynchronously was number 1234.

STATUS REPORTING

This shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work.

To check on the status of a process, use the ps (process status) command.

SUBSTITUTIONS

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions place words from previous command input as portions of new commands, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence.

History substitutions begin with the character '!' and may begin anywhere in the input stream (with the proviso that they do not nest.)

This '!' may be preceded by an '\' to turn off its special meaning; for convenience, a '!' is also passed unchanged when it is followed by a blank, tab, newline, '=' or '('.

Therefore, do _not_ put a space after the '!' and the command reference when you are invoking the shell's history mechanism. (History substitutions also occur when an input line begins with '↑'. This special abbreviation will be described later.)

An input line which invokes history substitution is echoed on the terminal before it is executed, as it would look if typed out in full.

The shell's history list, which may be seen by typing the "history" command, contains all commands input from the terminal which consist of one or more words. History substitutions reintroduce sequences of words from these saved commands into the input stream. The _history_ variable controls the size of the input stream. The previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

Consider the following output from the _history_ command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the _prompt_ by placing an '!' in the prompt string. This is done by SETting Prompt = ! and the prompt character of your choice.

For example, if the current event is number 13, we can call up the command recorded as event 11 in several ways: as !-2 [i.e., 13-2];

by the first letter of one of its command words, such as !c referring to the 'c' in _cat_;

or !wri for event 9, or by a string contained in a word in the command as in '!?mic?' also referring to event 9.

These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case '!!' refers to the previous command; thus '!!' alone is essentially a _redo_.

Words are selected from a command event and acted upon according to the following formula:

event:position:action

The "event" is the command you wish to retrieve. As mentioned above, it may be summoned up by event number and in several other ways. All that the "event" notation does is to tell the shell <u>which</u> command you have in mind.

"Position" picks out the words from the command event on which you want the "action" to take place. The "position" notation can do anything from altering the command completely to making some very minor substitution, depending on which words from the command event you specify with the "position" notation.

To select words from a command event, follow the event specification with a ':' and a designator (by position) for the desired words.

The words of a command event are picked out by their position in the input line. Positions are numbered from 0, the first word (usually command) being position 0, the second word having position 1, and so forth. If you designate a word from the command event by stating its position, that means you want to include it in your revised command. All the words that you want to include in a revised command must be designated by position notation in order to be included.

The basic position designators are:

0	first (command) word
<u>n</u>	<u>n</u>'th argument
↑	first argument, i.e. '1'
$	last argument
%	matches the word of an ?s? search which immediately precedes it; used to strip one word out of a command event for use in another command. Example: !?four?:%:p prints "four".

x-y   range of words (e.g. 1-3 means 'from position 1 to position 3').
-y   abbreviates '0-y'
*   stands for '↑-$', or indicates position 1 if only one word in event.
x*   abbreviates 'x-$' where x is a position number.
x-   like 'x*' but omitting last word '$'

The ':' separating the event specification from the word designator can be omitted if the argument selector begins with a '↑', '$', '*' "-" or "%".

Modifiers, each preceded by a ':', may be used to act on the designated words in the specified command event. The following modifiers are defined:

h   Remove a trailing pathname component, leaving the head.
r   Remove a trailing '.xxx' component, leaving the root name.
e   Remove all but the extension '.xxx' part.
s/old/new/ Substitute new for old
t   Remove all leading pathname components, leaving the tail.
&   Repeat the previous substitution.
g   Apply the change globally, prefixing the above, e.g. 'g&'.
p   Print the new command but do not execute it.
q   Quote the substituted words, preventing further substitutions.
x   Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a 'g' the modification is applied only to the first modifiable word. With substitutions, it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of '/'; a '\' quotes the delimiter into the l and r strings. The character '&' in the right hand side is replaced by the text from the left. A '\' quotes '&' also. A null l uses the previous string either from a l or from a contextual scan string s in '!?s?'. The trailing delimiter in the substitution may be omitted if (but only if) a newline follows immediately as may the trailing '?' in a contextual scan.

A history reference may be given without an event specification, e.g. '!$'. In this case the reference is to the previous command. If a previous history reference occurred on the same line, this form repeats the previous reference. Thus '!?foo?↑ !$' gives the first and last arguments from the command matching '?foo?'.

You can quickly make substitutions to the previous command line by using the '↑' character as the first non-blank character of an input line, This is equivalent to '!:s↑' providing a convenient shorthand for substitutions on the text of the previous line. Thus '↑lb↑lib' fixes the spelling of "lib" in the previous command. Finally, a history substitution may be surrounded with '{' and '}' if necessary to insulate it from the characters which follow. Thus, after 'ls -ld ~paul' we might do '!{l}a' to do 'ls -ld ~paula', while '!la' would look for a command starting 'la'.

Quotations with ' and "

The quotation of strings by ''' and '"' can be used to prevent all or some of the remaining substitutions which would otherwise take place if these characters were interpreted as "metacharacters" or "wild card matching characters". Strings enclosed in single quotes, ''' are prevented any further interpretation or expansion. Strings enclosed in '"' may still be variable and command expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see Command Substitution below) does a '"' quoted string yield parts of more than one word; "'" quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the alias and unalias commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for 'ls' is 'ls -l' the command 'ls /usr' would map to "ls -l /usr", the argument list here being undisturbed. Similarly if the alias for 'lookup' was 'grep !↑ /etc/passwd' then "lookup bill" would map to "grep bill /etc/passwd".

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can 'alias print 'pr \!* | lpr'' to make a command which pr's its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the argv variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle which causes command input to be echoed. The setting of this variable results from the -v command line option.

Other operations treat variables numerically. The '@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by '$' characters. This expansion can be prevented by preceding the '$' with a '\' except within '"'s where it always occurs, and within '''s where it never occurs. Strings quoted by '`' are interpreted later (see *Command substitution* below) so '$' substitution does not occur there until later, if at all. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.

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

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

Metasequences for variable substitution

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

$name
${name}

Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter.
If *name* is not a shell variable, but is set in the environment,

then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}
May be used to select only some of the words from the value of name. The selector is subjected to ´$´ substitution and may consist of a single number or two numbers separated by a ´-´. The first word of a variables value is numbered ´1´. If the first number of a range is omitted it defaults to ´1´. If the last member of a range is omitted it defaults to ´$#name´. The selector ´*´ selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}
Gives the number of words in the variable. This is useful for later use in a ´[selector]´.

$0
Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}
Equivalent to ´$argv[number]´.

$*
Equivalent to ´$argv[*]´.

The modifiers ´:h´, ´:t´, ´:r´, ´:q´ and ´:x´ may be applied to the substitutions above as may ´:gh´, ´:gt´ and ´:gr´. If braces ´{´ ´}´ appear in the command form then the modifiers must appear within the braces. The current implementation allows only one ´:´ modifier on each ´$´ expansion.

The following substitutions may not be modified with ´:´ modifiers.

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

$?0
Substitutes ´1´ if the current input filename is know, ´0´ if it is not.

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

$<
Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the

keyboard in a shell script.

Command and filename substitution

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

Command substitution

Command substitution is indicated by a command enclosed in '`'. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within '"'s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters '*', '?', '[' or '{' or begins with the character '~', then that word is a candidate for filename substitution, also known as 'globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters '*', '?' and '[' imply pattern matching, the characters '~' and '{' being more akin to abbreviations.

In matching filenames, the character '.' at the beginning of a filename or immediately following a '/', as well as the character '/' must be matched explicitly. The character '*' matches any string of characters, including the null string. The character '?' matches any single character. The sequence '[...]' matches any one of the characters enclosed. Within '[...]', a pair of characters separated by '-' matches any character lexically between the two.

The character '~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. '~' it expands to the invokers home directory as reflected in the value of the variable <u>home</u>. When followed by a name consisting of letters, digits and '-' characters the shell searches for a user with that name and substitutes their home directory; thus '~ken' might expand to '/usr/ken' and '~ken/chmach' to '/usr/ken/chmach'. If the character '~' is followed by a character other than a letter or '/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation 'a{b,c,d}e' is a shorthand for 'abe ace ade'. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus '~source/s1/{oldls,ls}.c' expands to "/usr/source/s1/oldls.c /usr/source/s1/ls.c" whether or not these files exist without any chance of error if the home directory for 'source' is '/usr/source'. Similarly '../{memo,*box}' might expand to '../memo ../box ../mbox'. (Note that 'memo' was not sorted with the results of matching '*box'.) As a special case '{', '}' and '{}' are passed undisturbed.

Input/output

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

< name

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

<< word

Read the shell input up to a line which is identical to *word*. *Word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting '\', '"', ''' or '`' appears in *word* variable and command substitution is performed on the intervening lines, allowing '\' to quote '$', '\' and '`'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

\> name
\>! name
\>& name
\>&! name

The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, it is truncated, its previous contents being lost.

If the variable *noclobber* is set, then the file must not exist or be a character special file (e.g. a terminal or '/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the '!' forms can be used and suppress this check.

The forms involving '&' route the diagnostic output into the specified file as well as the standard output. *Name* is expanded in the same way as '<' input filenames are.

\>> name
\>>& name
\>>! name
\>>&! name

Uses file *name* as standard output like '>' but places output at the end of the file. If the variable *noclobber* is set, then it is an error for the file not to exist unless one of the '!' forms is given. Otherwise similar to '>'.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The '<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form '|&' rather than just '|'.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, *exit*, *if*, and *while* commands. The following operators are available:

```
    ||  &&  |  ↑  &  ==  !=  =~  !~  <=  >=  <  >  <<  >>  +  -  *  /
%  !  ~  (  )
```

Here the precedence increases to the right, "==" "!=" "=~" and "!~", "<=" ">=" "<" and ">", "<<" and ">>", "+" and "-", "*" "/" and "%" being, in groups, at the same level. The "==" "!=" "=~" and "!~" operators compare their arguments as strings; all others operate on numbers. The operators '=~' and '!~' are like '!=' and '==' except that the right hand side is a *pattern* (containing, e.g. '*'s, '?'s and instances of '[...]') against which the left hand operand is matched. This reduces the need for use of the *switch* statement in shell scripts when all that is really needed is pattern matching.

Strings which begin with '0' are considered octal numbers. Null or missing arguments are considered '0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('&' '|' '<' '>' '(' ')') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in '{' and '}' and file enquiries of the form '-*l* name' where *l* is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, then all enquiries return false, i.e. '0'. Command executions succeed, returning true, i.e. '1', if the command exits with status 0, otherwise they fail, returning false, i.e. '0'. If more detailed status information is required then the command should be executed outside of an expression and the variable _status_ examined.

CONTROL FLOW

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The _foreach_, _switch_, and _while_ statements, as well as the _if_-_then_-_else_ form of the _if_ statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

BUILTIN COMMANDS

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias name
alias name wordlist

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified _wordlist_ as the alias of _name_; _wordlist_ is command and filename substituted. _Name_ is not allowed to be _alias_ or _unalias_.

break

Causes execution to resume after the _end_ of the nearest enclosing _foreach_ or _while_. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a _switch_, resuming after the _endsw_.

case label:
A label in a _switch_ statement as discussed below.

cd
cd name
chdir
chdir name
Change the shells working directory to directory _name_. If no argument is given then change to the home directory of the user.
If _name_ is not found as a subdirectory of the current directory (and does not begin with '/', './' or '../'), then each component of the variable _cdpath_ is checked to see if it has a subdirectory _name_. Finally, if all else fails but _name_ is a shell variable whose value begins with '/', then this is tried to see if it is a directory.

continue
Continue execution of the nearest enclosing _while_ or _foreach_. The rest of the commands on the current line are executed.

default:
Labels the default case in a _switch_ statement. The default should come after all _case_ labels.

echo wordlist
echo -n wordlist
The specified words are written to the shells standard output, separated by spaces, and terminated with a newline unless the -n option is specified.

else
end
endif
endsw
See the description of the _foreach_, _if_, _switch_, and _while_ statements below.

exec command
The specified command is executed in place of the current shell.

exit
exit(expr)
The shell exits either with the value of the _status_ variable (first form) or with the value of the specified _expr_ (second form).

foreach name (wordlist)
...
end
The variable _name_ is successively set to each member of _wordlist_

and the sequence of commands between this command and the matching _end_ are executed. (Both _foreach_ and _end_ must appear alone on separate lines.)

The builtin command _continue_ may be used to continue the loop prematurely and the builtin command _break_ to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with '?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob wordlist
: Like _echo_ but no '\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word
: The specified _word_ is filename and command expanded to yield a string of the form 'label'. The shell rewinds its input as much as possible and searches for a line of the form 'label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

history
: Displays the history event list.

if (expr) command
: If the specified expression evaluates true, then the single _command_ with arguments is executed. Variable substitution on _command_ happens early, at the same time it does for the rest of the _if_ command. _Command_ must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if _expr_ is false, when command is not executed (this is a bug).

```
if (expr) then
    ...
else if (expr2) then
    ...
else
    ...
endif
```

If the specified _expr_ is true then the commands to the first _else_ are executed; else if _expr2_ is true then the commands to the second else are executed, etc. Any number of _else-if_ pairs are possible; only one _endif_ is needed. The _else_ part is likewise optional. (The words _else_ and _endif_ must appear at the beginning of input lines; the _if_ must appear alone on its input line or after an _else_.)

kill pid

kill -sig pid ...
kill -l

Sends either the TERM (terminate) signal or the specified signal to the specified processes. Signals are either given by number or by names (as given in /*usr*/*include*/*signal*.*h*, stripped of the prefix "SIG"). The signal names are listed by "kill -l". There is no default, saying just 'kill' does not send a signal to the current process. If the signal being sent is TERM (terminate) or HUP (hangup), then the job or process will be sent a CONT (continue) signal as well.

login

Terminate a login shell, replacing it with an instance of /bin/login. This is one way to log off, included for compatibility with *sh*(1).

logout

Terminate a login shell. Especially useful if *ignoreeof* is set.

nice
nice +number
nice command
nice +number command

The first form sets the *nice* for this shell to 4. The second form sets the *nice* to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user may specify negative niceness by using 'nice -number ...'. Command is always executed in a sub-shell, and the restrictions place on commands in simple *if* statements apply.

nohup
nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. All processes detached with '&' are effectively *nohup'ed*.

onintr
onintr -
onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form 'onintr -' causes all interrupts to be ignored. The final form causes the shell to execute a 'goto label' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

rehash

Causes the internal hash table of the contents of the directories in the _path_ variable to be recomputed. This is needed if new commands are added to directories in the _path_ while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command

The specified _command_ which is subject to the same restrictions as the _command_ in the one line _if_ statement above, is executed _count_ times. I/O redirections occur exactly once, even if _count_ is 0.

set
set name
set name=word
set name[index]=word
set name=(wordlist)

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets _name_ to the null string. The third form sets _name_ to the single _word_. The fourth form sets the _index'th_ component of name to word; this component must already exist. The final form sets _name_ to the list of words in _wordlist_. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv name value

Sets the value of environment variable _name_ to be _value_, a single string. The variable PATH is automatically imported to and exported from the _csh_ variable _path_; there is no need to use _setenv_ for these.

shift
shift variable

The members of _argv_ are shifted to the left, discarding _argv_[_1_]. It is an error for _argv_ not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name

The shell reads commands from _name_. _Source_ commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a _source_ at any level terminates all nested _source_ commands. Input during _source_ commands is never placed on the history list.

```
switch (string)
case str1:
    ...
  breaksw
...
default:
    ...
  breaksw
endsw
```

Each case label is successively matched against the specified _string_ which is first command and filename expanded. The file metacharacters '*', '?' and '[...]' may be used in the case labels, which are variable expanded. If none of the labels match before a 'default' label is found, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command _breaksw_ causes execution to continue after the _endsw_. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the _endsw_.

time
time command

With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the _time_ variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by 'unalias *'. It is not an error for nothing to be _unaliased_.

unhash

Use of the internal hash table to speed location of executed programs is disabled.

unset pattern

All variables whose names match the specified pattern are removed. Thus all variables are removed by 'unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be _unset_.

wait

All background jobs are waited for. If the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

while (expr)
...
end

While the specified expression evaluates non-zero, the commands between the _while_ and the matching end are evaluated. _Break_ and _continue_ may be used to terminate or continue the loop prematurely. (The _while_ and _end_ must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the _foreach_ statement if the input is a terminal.

@
@ name = expr
@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified _name_ to the value of _expr_. If the expression contains '<', '>', '&' or '|' then at least this part of the expression must be placed within '(' ')'. The third form assigns the value of _expr_ to the _index'th_ argument of _name_. Both _name_ and its _index'th_ component must already exist.

The operators '*=', '+=', etc are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of _expr_ which would otherwise be single words.

Special postfix '++' and '--' operators increment and decrement _name_ respectively, i.e. '@ i++'.

PRE-DEFINED AND ENVIRONMENT VARIABLES

The following variables have special meaning to the shell. Of these, _argv_, _home_, _path_, _prompt_, _shell_ and _status_ are always set by the shell. Except for _status_ this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

This shell copies the environment variable USER into the variable _user_, TERM into _term_, and HOME into _home_, and copies these back into the environment whenever the normal shell variables are reset. The environment variable PATH is likewise handled; it is not necessary to worry about its setting other than in the file _.cshrc_ as inferior _csh_ processes will import the definition of _path_ from the environment, and re-export it if you then change it.

argv

Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. '$1' is replaced by "$argv[1]", etc.

cdpath
: Gives a list of alternate directories searched to find subdirectories in _chdir_ commands.

echo
: Set when the -x command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

history
: Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of _history_ may run the shell out of memory. The last executed command is always saved on the history list.

home
: The home directory of the invoker, initialized from the environment. The filename expansion of ´~´ refers to this variable.

ignoreeof
: If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D´s.

mail
: The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says ´You have new mail.´ if the file exists with an access time not greater than its modify time.

  If the first word of the value of _mail_ is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

  If multiple mail files are specified, then the shell says "New mail in _name_´ when there is mail in the file _name_.

noclobber
: As described in the section on _Input_/_output_, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that ´>>´ redirections refer to existing files.

noglob
: If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.

nonomatch
: If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. "echo [" still gives an

error.

path — Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no _path_ variable then only full path names will execute. The usual search path is '.', '/bin' and '/usr/bin', but this may vary from system to system. For the super-user the default search path is '/etc', '/bin' and '/usr/bin'. A shell which is given neither the -c nor the -t option will normally hash the contents of the directories in the _path_ variable after reading _.cshrc_, and each time the _path_ variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the _rehash_ or the commands may not be found.

prompt — The string which is printed before each command is read from an interactive terminal input. If a '!' appears in the string it will be replaced by the current event number unless a preceding '\' is given. Default is '% ', or '# ' for the super-user.

shell — The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of _Non-builtin Command Execution_ below.) Initialized to the (system-dependent) home of the shell.

status — The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status '1', all other builtin commands set status '0'.

time — Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose — Set by the -v command line option, causes the words of each command to be printed after history substitution.

NON-BUILTIN COMMAND EXECUTION

When a command to be executed is found not to be a builtin command the shell attempts to execute the command via _exec_(2). Each word in the variable _path_ names a directory from which the shell will attempt to execute the command. If it is given neither a -c nor a -t option, the shell will hash the names in these directories into an internal table so that it will only try an _exec_ in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via _unhash_), or if the shell was given a -c or -t argument, and in any case for each directory component of _path_ which does not begin with a '/', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus "(cd ; pwd) ; pwd" prints the _home_ directory; leaving you where you were (printing this after the home directory), while "cd ; pwd" leaves you in the _home_ directory. Parenthesized commands are most often used to prevent _chdir_ from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands an a new shell is spawned to read it.

If there is an _alias_ for _shell_ then the words of the alias will be prepended to the argument list to form the shell command. The first word of the _alias_ should be the full path name of the shell (e.g. "$shell"). Note that this is a special, late occurring, case of _alias_ substitution, and only allows words to be prepended to the argument list without modification.

ARGUMENT LIST PROCESSING

If argument 0 to the shell is '-' then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in _argv_.

-e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell will start faster, because it will neither search for nor execute commands from the file ".cshrc" in the invokers home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A '\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the _verbose_ variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the _echo_ variable to be set, so that commands are echoed immediately before execution.

-V Causes the _verbose_ variable to be set even before '.cshrc' is executed.

-X Is to -x as -V is to -v.

After processing of flag arguments, if arguments remain but none of the -c, -i, -s, or -t options was given, the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '$0'. Remaining arguments initialize the variable _argv_.

SIGNAL HANDLING

The shell normally ignores _quit_ signals. Processes running in background (by '&') are immune to signals generated from the keyboard, including hangups. Other signals have the values which the shell inherited from its parent. The shells handling of interrupts and terminate signals in shell scripts can be controlled by _onintr_. Login shells catch the _terminate_ signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file ".logout".

AUTHOR

William Joy.

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after '.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Standard shell, for shell scripts not starting with a '#'.
/tmp/sh*	Temporary file for '<<'.
/etc/passwd	Source of home directories for '~name'.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 5120 characters. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an argument list. To detect looping, the shell restricts the number of _alias_ substitutions on a single line to 20.

SEE ALSO

sh(1), access(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), tty(4), a.out(5), environ(5),
and especially, "An introduction to the C shell" by William Joy.

BUGS

It suffices to place the sequence of commands in ()'s to force it to a subshell, i.e. '(a ; b ; c)'.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by '?', are not placed in the _history_ list. Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with '|', and to be used with '&' and ';' metasyntax.

It should be possible to use the ':' modifiers on the output of command substitutions. All and more than one ':' modifier should be allowed on '$' substitutions.

NAME

ctags - maintain a tags file for a C program

SYNOPSIS

ctags [-a] [-u] [-w] [-x] name ...

DESCRIPTION

Ctags makes a tags file for *ex*(1) and *vi*(1) from the specified C, Fortran, and Pascal sources.

A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these function definitions.

OPTIONS

The -*a* option causes the output to be appended to the tags file instead of rewriting it.

The -*u* option causes the specified files to be *updated* in tags, that is, all references to them are deleted, and the new values are appended to the file. This option implies the -*a* option. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

The -*w* option suppresses warning diagnostics.

If the -x flag is given, *ctags* produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output.

Files whose name ends in .c or .h are assumed to be C source files and are searched for C routine and macro definitions.

The tag *main* is treated specially in C programs. The tag formed is created by prepending *M* to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

EXAMPLE

```
ctags *.c *.h
```

puts the tags from all the ".c" and ".h" files into the tagsfile *tags*.

FILES

tags output tags file

SEE ALSO

ex(1), vi(1)

AUTHOR

Ken Arnold

NAME

cu - call UNIX

SYNOPSIS

cu telno [-t] [-*n* [-s speed] [-a acu] [-l line] [-b]

DESCRIPTION

Cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of text files. *Telno* is the telephone number, with minus signs at appropriate places for delays. The -t flag is used to dial out to a terminal. *Speed* gives the transmission speed (110, 134, 150, 300, 1200); 300 is the default value.

The -a and -l values may be used to specify pathnames for the ACU and communications line devices. They can be used to override the following built-in choices:

-a /dev/cua0 -l /dev/cul0

The -*n* option, where *n* is a single digit, changes the last character of the ACU and communications line to *n*. It is an abbreviation for -a /dev/cua*n* -l /dev/cul*n*.

After making the connection, *cu* runs as two processes: the *send* process reads the standard input and passes most of it to the remote system; the *receive* process reads from the remote system and passes most data to the standard output. Lines beginning with `~' have special meanings.

The *send* process interprets the following:

~.	terminate the conversation.
~EOT	terminate the conversation
~<file	send the contents of *file* to the remote system, as though typed at the terminal.
~^Z	suspend the cu process. Note that the control-Z must be followed by a newline.
~#	sends a break.
~!	invoke an interactive shell on the local system.
~!cmd ...	run the command on the local system (via sh -c).
~$cmd ...	run the command locally and send its output to the remote system.
~%take from [to]	copy file `from' (on the remote system) to file `to' on the local system. If `to' is omitted, the `from' name is used both places.

~%put from [to] copy file `from´ (on local system) to file `to´ on remote system. If `to´ is omitted, the `from´ name is used both places.

~: during an output diversion, this toggles whether the operation of *cu* will be silent, i.e., whether information received from the foreign system will be written to the standard output. This allows a ``progress report´´ during long transfers.

~~... send the line `~...´.

Both the *send* and *receive* processes handles output diversions of the following form:

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

In any case, output is diverted (or appended, if `>>´ used) to the file. If `:´ is used, the diversion is *silent*, i.e., it is written only to the file. If `:´ is omitted, output is written both to the file and to the standard output. The trailing `~>´ terminates the diversion.

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

The use of ~%take requires the existence of *echo* and *tee* on the remote system. Also, stty tabs mode is required on the remote system if tabs are to be copied without expansion.

Finally, the -b flag specifies that nulls are to be turned into breaks. This allows the break key (and also control-shift-@) to send a break.

FILES

/dev/cua0
/dev/cul0
/dev/null
/usr/spool/uucp/LCK..cu[al][0-7]

SEE ALSO

rv(4), tty(4)

DIAGNOSTICS

Exit code is zero for normal exit, nonzero (various values) otherwise.

BUGS

Only *mail*(1) uses syntax anything like the syntax of *cu*.

NAME

date - print and set the date

SYNOPSIS

date [yy[mm[dd[hh[mm[.ss]]]]]]

DESCRIPTION

If no argument is given, the current date and time are printed. If an argument is given, the current date is set. _yy_ is the last two digits of the year; the first _mm_ is the month number; _dd_ is the day number in the month; _hh_ is the hour number (24 hour system); the second _mm_ is the minute number; ._ss_ is optional and is the seconds.

EXAMPLE

```
date 10080045
```

sets the date to Oct 8, 12:45 AM. The year, month and day may be omitted, the current values being the defaults. The system operates in GMT (Greenwich Mean Time). _Date_ takes care of the conversion to and from local standard and daylight time.

FILES

/usr/adm/wtmp to record time-setting

DIAGNOSTICS

"No permission" if you aren't the super-user and you try to change the date; "bad conversion" if you are the super-user but the date set is syntactically incorrect.

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 numbering system such as base 8 or base 16, an 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. Negative numbers for input are indicated by being immediately preceded by an underscore _. Numbers may contain decimal points.

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

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

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

d
: The top value on the stack is duplicated.

p
: The top value on the stack is printed. The top value remains unchanged. P interprets the top of the stack as an ascii string, removes it, and prints it.

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

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

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

X replaces the number on the top of the stack with its scale factor.

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

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

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

! interprets the rest of the line as a 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. I pushes the input base on the top of the stack.

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

O pushes the output base on the top of the stack.

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

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

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

EXAMPLES

```
dc
24.2 56.2 + p
```

adds the two numbers and prints the result (top value in the stack).

To exit from *dc*, hit control-d (EOF).

DIAGNOSTICS

"x is unimplemented" where x is an octal number.
"stack empty" for not enough elements on the stack to do what was asked.
"Out of space" when the free list is exhausted (too many digits).

"Out of headers" for too many numbers being kept around.
"Out of pushdown" for too many items on the stack.
"Nesting Depth" for too many levels of nested execution.

NAME

dcheck - file system directory consistency check

SYNOPSIS

dcheck [-i numbers] [filesystem]

DESCRIPTION

N.B.: _Dcheck_ has been made obsolete for normal consistency checking by _fsck_(1M).

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.

EXAMPLE

```
dcheck /dev/rdiskl
```

checks the consistency of the device _rdiskl_.

FILES

Default file systems vary with installation.

SEE ALSO

fsck(1M), icheck(1M), filsys(5), clri(1M), ncheck(1M)

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.

BUGS

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

Dcheck has been superseded by _fsck_ and remains for historical reasons.

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=*n*	input block size *n* bytes (default 512)
obs=*n*	output block size (default 512)
bs=*n*	set both input and output block size, superseding *ibs* and *obs*; also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
files=*n*	skip *n* input files before starting copy
seek=*n*	seek *n* records from beginning of output file before copying
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
block	convert variable length records to fixed length
unblock	convert fixed length records to variable length
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to *ibs*
... , ...	several comma-separated conversions

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

Cbs is used only if *ascii*, *unblock*, *ebcdic*, *ibm*, or *block* conversion is specified. In the first two cases, *cbs* characters are placed into the conversion buffer, any specified character mapping is done, trailing blanks trimmed and new-line added before sending the line to the output. In the latter three cases, characters are read into the conversion buffer, 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.

EXAMPLES

```
dd if=filename conv=ucase
```

changes the alphabetics in the input file _file_ to upper case and writes to the standard output.

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

reads an EBCDIC tape blocked ten 80-byte card images per record into the ASCII file _x_. 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(1), tr(1)

DIAGNOSTICS
f+p records in(out): numbers of full and partial records (blocks) read/written

BUGS
The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. The 'ibm' conversion, while less blessed as a standard, corresponds better to certain IBM print train conventions. There is no universal solution.

NAME

delta - make a delta (change) to an SCCS file

SYNOPSIS

delta [-rSID] [-s] [-n] [-glist] [-m[mrlist]] [-y[comment]] [-p] files

DESCRIPTION

Delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(1) (called the *g-file*, or generated file).

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

Delta may issue prompts on the standard output depending upon certain keyletters specified and flags (see *admin*(1)) that may be present in the SCCS file (see -m and -y keyletters below).

Keyletter arguments apply independently to each named file.

-r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding *get*s for editing (get -e) on the same SCCS file were done by the same person (login name). The SID value specified with the -r keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(1)). A diagnostic results if the specified SID is ambiguous, or, if necessary and omitted on the command line.

-s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

-n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

-g*list* Specifies a *list* (see *get*(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

-m[*mrlist*] If the SCCS file has the v flag set (see *admin*(1)) then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

If -m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see -y keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the v flag has a value (see _admin_(1)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, _delta_ terminates (it is assumed that the MR numbers were not all valid).

-y[_comment_] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid _comment_.

If -y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped new-line character terminates the comment text.

-p Causes _delta_ to print (on the standard output) the SCCS file differences before and after the delta is applied in a _diff_(1) format.

FILES

All files of the form ?-file are explained in the _Source Code Control System User's Guide_. The naming convention for these files is also described there.

g-file	Existed before the execution of _delta_; removed after completion of _delta_.
p-file	Existed before the execution of _delta_; may exist after completion of _delta_.
q-file	Created during the execution of _delta_; removed after completion of _delta_.
x-file	Created during the execution of _delta_; renamed to SCCS file after completion of _delta_.
z-file	Created during the execution of _delta_; removed during the execution of _delta_.
d-file	Created during the execution of _delta_; removed after completion of _delta_.
/usr/bin/bdiff	Program to compute differences between the "gotten"

file and the _g-file_.

WARNINGS

Lines beginning with an SOH ASCII character (binary 001) cannot be placed in the SCCS file unless the SOH is escaped. This character has special meaning to SCCS (see _sccsfile_(5)) and will cause an error.

A _get_ of many SCCS files, followed by a _delta_ of those files, should be avoided when the _get_ generates a large amount of data. Instead, multiple _get_/_delta_ sequences should be used.

If the standard input (-) is specified on the _delta_ command line, the -m (if necessary) and -y keyletters _must_ also be present. Omission of these keyletters causes an error to occur.

SEE ALSO

admin(1), bdiff(1), get(1), help(1), prs(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use _help_(1) for explanations.

NAME

deroff - remove nroff, troff, tbl and eqn constructs

SYNOPSIS

deroff [-w] file ...

DESCRIPTION

Deroff reads each file in sequence and removes all _nroff_ and _troff_ command lines, backslash constructions, macro definitions, _eqn_ constructs (between '.EQ' and '.EN' lines or between delimiters), and table descriptions and writes the remainder on the standard output.

Deroff follows chains of included files ('.so' and '.nx' commands); if a file has already been included, a '.so' is ignored and a '.nx' terminates execution. If no input file is given, _deroff_ reads from the standard input file.

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

EXAMPLE

```
deroff textfile
```

Removes all _nroff_, _troff_, and macro definitions from _textfile_.

SEE ALSO

troff(1), eqn(1), tbl(1)

BUGS

Deroff is not a complete _troff_ interpreter, so it can be confused by subtle constructs. Most errors result in too much rather than too little output.

NAME

df - disk free

SYNOPSIS

df [filesystem ...] [file ...]

DESCRIPTION

_Df_ prints out the number of free blocks available on the specified _filesystem_, e.g. "/dev/rw0a". If no file system is specified, the free space on all of the mounted file systems plus the systems listed in /etc/checklist are printed.

The reported numbers are in file system block units. Each filesystem block is 512 bytes long.

EXAMPLE

```
df /dev/rw0a
```

would report the number of free disk blocks [512 bytes each] on /dev/rw0a.

FILES

/etc/mtab	list of currently mounted filesystems
/etc/checklist	list of normally mounted filesystems

SEE ALSO

icheck(1M)

NAME

diff - differential file comparator

SYNOPSIS

diff [-efb] file1 file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If either one of the *files* is represented by '-', the standard input is used.

Moreover, one of the file names could be that of a directory: in this case the comparison is between two files of the same name. Either the file or the directory can be named first for the *diff*, but the directory must be a sub-directory of *file's* directory (i.e. below it in the tree structure).

The output from a *diff* produces lines of these forms:

n1 a *n3*,*n4*
n1,*n2* d *n3*
n1,*n2* c *n3*,*n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

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

The -b option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The -e option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The -f option produces a similar script, not useful with *ed*, in the opposite order. In connection with -e, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A 'latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

EXAMPLE

```
diff -e file1 file2
```

where file1 and file2 are two versions of the manual text for the _cp_ command, produces:

```
35,41d
27c
In the second form, one or more

18,25c
existed; the mode of the source file
is used otherwise.

15c
The mode and owner of

10c
file ... directory

7c
file1 file2

1,3c
 .TH CP 1
 .SH NAME
```

Following this _ed_ script would transform _file1_ into _file2_, line for line and character for character.

FILES

/usr/lib/diffh to compare big files

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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

NAME

diffdir - diff directories

SYNOPSIS

diffdir [-h] [-s] dirl dir2

DESCRIPTION

Diffdir does diffs on directories recursively by sorting the contents of the directories by name and then running diffs on text files which are different. Object files which differ and files which appear in only one directory are also listed.

The -h option causes diffdir to paginate its output, and to summarize binary differences and files in only one place at the end of the diff. Each individual diff is run through an appropriate *pr*.

The -s option causes files which are the same to be reported; normally they are omitted.

EXAMPLES

```
diffdir dirl dir2
```

compares all the files in two directories and reports differences, by line number, for similar files. Unique files are simply listed.

FILES

/usr/bin/cmp compare two files

SEE ALSO

diff(1)

AUTHOR

Bill Joy

BUGS

Program should pass flags through to *diff*.

NAME

diskformat - format a disk

SYNOPSIS

diskformat [-size #] [-dens #] [-cyl f[-t]] [-sec f[-t]] [-i] #] device

DESCRIPTION

Diskformat initializes a hard disk or floppy disk and formats it according to your specifications.

The following parameters may be specified ("device" is required):

device
: device to be formatted (must be raw device)

-size #
: specify sector size in bytes

-dens #
: specify density

-cyl #[-#]
: format cylinders f to t (default f). A specification such as #- means "until the end".

-head #[-#]
: Format heads f to t (default f). A specification such as #- means "until the end".

-sec #[-#]
: Format sectors f to t (default f). A specification such as #- means "until the end".

-il # Interleave factor for the disk.

EXAMPLE

```
diskformat /dev/rfdc0 -dens 1 -size 128 -il 3
```

will format the floppy disk on drive 0, single density, 128 bytes per sector with an interleave factor of 3. This format is the only truly portable floppy format.

NAME

disktune - tune the floppy disk settling time parameters

SYNOPSIS

disktune [-srt #] [-hlt #] [-hut #] device

DESCRIPTION

Disktune tunes the floppy disk settling time parameters. These include the motor stepping rate and the rate at which the head loads and unloads. _Disktune_ thus enables you to obtain the most efficient operation from your floppy disk.

If no settable parameters are given, _disktune_ will report the current settings on _device_. _Disktune_ retains the current settings on parameters which are not specified.

The settable parameters are:

-srt #
: seek motor stepping rate time in ms

-hlt #
: head loading time in ms

-hut #
: head unload time in ms

EXAMPLE

```
disktune -srt 3 /dev/rfdc0
```

will set the step rate time on the floppy controller to 3 ms per step.

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, "." the current directory 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.

EXAMPLE

```
du dirl dir2
```

produces a count of the number of blocks in each of the directories.

In order to see how many blocks are in each file, the -a option must be used.

SEE ALSO

df(1)

BUGS

Non-directories given as arguments (not under -a option) are not listed. If there are too many distinct linked files, *du* counts the excess files more than once.

NAME

dump - incremental file system dump

SYNOPSIS

dump [key [_argument_ ...] filesystem]

DESCRIPTION

Dump copies to tape or disk all files changed after a certain date in the _filesystem_. The _key_ specifies the date and other options about the dump. _Key_ consists of characters from the set 0123456789bfusdn.

0-9 This number is the 'dump level'. All files modified since the last date stored in the file /_etc_/_ddate_ for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option 0 causes the entire filesystem to be dumped.

f Place the dump on the next _argument_ file or dump device [such as a floppy or hard disk] instead of the default tape.

b Specifies the number of blocks on the dump device. Used to specify the number of blocks floppy disks will hold, so that the dump will pause while disks are changed.

u If the dump completes successfully, write the date of the beginning of the dump on file /_etc_/_ddate_. This file records a separate date for each filesystem and each dump level.

s The size of a dump tape is specified in feet. The number of feet is taken from the next _argument_. When the specified size is reached, _dump_ will wait for reels to be changed. The default tape size is 2300 feet.

d The density of the tape, expressed in BPI, is taken from the next _argument_. This is used in calculating the amount of tape used per reel. The default is 1600.

If no arguments are given, the _key_ is assumed to be 9u and a default file system is dumped to the default tape.

Dump requires operator intervention on these conditions: end of disk or tape, end of dump, disk write error, disk or tape open error or read error.

Dump interacts with the operator on _dump's_ control terminal at times when _dump_ can no longer proceed, or if something is grossly wrong. All questions _dump_ poses must be answered by typing yes or no, appropriately.

Now a short suggestion on how to perform dumps. Start with a full level 0 dump

dump 0u

Next, dumps of active file systems are taken on a daily basis, using a modified Tower of Hanoi algorithm, with this sequence of dump levels:

3 2 5 4 7 6 9 8 9 9 ...

For the daily dumps, a set of 10 sets of disks or tapes per dumped file system is used on a cyclical basis. Each week, a level 1 dump is taken, and the daily Hanoi sequence repeats with 3. For weekly dumps, a set of 5 sets of disks or tapes per dumped file system is used, also on a cyclical basis. Each month, a level 0 dump is taken on a set of fresh disks or tapes that is saved forever.

EXAMPLE

```
dump  0bf 2310 /dev/rfdc0 /dev/rmsc0a
```

would perform a level "0" dump to the floppy disk device rfdc0, which has 2310 blocks. The filesystem to be dumped is /dev/rmsc0a. Note that all the parameters in the key are grouped first in the command line, followed by the dump device (if other than tape), size etc. The last argument should be the pathname of the file system being dumped.

FILES

/dev/rmt1	default tape unit to dump to
/dev/rrp3	default disk unit to dump from
/etc/ddate	dump date record

SEE ALSO

restor(1M), dump(5), dumpdir(1M)

DIAGNOSTICS

Many, and verbose.

BUGS

Sizes are based on 1600 BPI blocked tape; the raw magtape device has to be used to approach these densities.

It would be nice if _dump_ knew about the dump sequence, kept track of the tapes scribbled on, told the operator which tape to mount when, and provided more assistance for the operator running _restor_.

NAME

dumpdir - print the names of files on a dump tape or disk

SYNOPSIS

/etc/dumpdir [f filename]

DESCRIPTION

Dumpdir is used to read magtapes or disks dumped with the *dump* command *Dumpdir* lists the names and inode numbers of all the files and directories on the backup tape or disk.

The f option causes *filename* as the name of the dump device, whether tape or disk.

FILES

default backup unit varies with installation
rst*

SEE ALSO

dump(1M), restor(1M)

DIAGNOSTICS

If the dump extends over more than one tape or disk, it will ask you to change tapes or disks. Reply with a new-line after the next one has been mounted.

NAME

echo - echo arguments

SYNOPSIS

echo [-n] [arg] ...

DESCRIPTION

Echo writes its arguments (separated by blanks and terminated by a newline) on the standard output. If the flag -n is used, no newline is added to the output.

Echo is useful for producing diagnostics in shell programs and for writing constant data on pipes.

To send diagnostics to the standard error file, do

```
echo ... 1 >& 2
```

in *sh*.

EXAMPLE

echo curmudgeon

simply responds

curmudgeon

on the standard output.

NAME

ed - text editor

SYNOPSIS

ed [-] [-p[prompt]] [-u] [-x] [name]

DESCRIPTION

Ed is the standard text editor.

If a *name* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed´s* buffer so that it can be edited. If -p is present, *ed* prompts for commands with ´* ´ (or *prompt* if given.) If -u is present, all lower case text in the buffer is converted to upper case. If -x is present, an *x* command is simulated first to handle an encrypted file. The optional - suppresses the printing of explanatory output and should be used when the standard input is an editor script.

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

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

In general, only one command may appear on a line. Certain commands allow the addition of text to the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period ´.´ alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. In the following specification for regular expressions the word ´character´ means any character but newline.

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus \[. and sometimes ^*$.

2. A . matches any character.

3. A \ followed by any character except a digit or () matches that character.

4. A nonempty string *s* bracketed [*s*] (or [^*s*]) matches any character in (or not in) *s*. In *s*, \ has no special meaning, and] may only appear as the first letter. A substring *a*-*b*, with *a* and *b* in

ascending ASCII order, stands for the inclusive range of ASCII characters.

5. A regular expression of form 1-4 followed by * matches a sequence of 0 or more matches of the regular expression.

6. A regular expression, _x_, of form 1-8, bracketed \(_x_\) matches what _x_ matches.

7. A \ followed by a digit _n_ matches a copy of the string that the bracketed regular expression beginning with the _n_th \(matched.

8. A regular expression of form 1-8, _x_, followed by a regular expression of form 1-7, _y_ matches a match for _x_ followed by a match for _y_, with the _x_ match being as long as possible while still permitting a _y_ match.

9. A regular expression of form 1-8 preceded by ^ (or followed by $), is constrained to matches that begin at the left (or end at the right) end of a line.

10. A regular expression of form 1-9 picks out the longest among the leftmost matches in a line.

11. An empty regular expression stands for a copy of the last regular expression encountered.

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

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

1. The character '.' addresses the current line.

2. The character '$' addresses the last line of the buffer.

3. A decimal number _n_ addresses the _n_-th line of the buffer.

4. "'_x_" addresses the line marked with the name _x_, which must be a lower-case letter. Lines are marked with the _k_ command described below.

5. A regular expression enclosed in slashes '/' addresses the line found by searching forward from the current line and stopping at

the first line containing a string that matches the regular expression. If necessary the search wraps around to the beginning of the buffer.

6. A regular expression enclosed in queries '?' addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the end of the buffer.

7. An address followed by a plus sign '+' or a minus sign '-' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.

8. If an address begins with '+' or '-' the addition or subtraction is taken with respect to the current line; e.g. '-5' is understood to mean '.-5'.

9. If an address ends with '+' or '-', then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address '-' refers to the line before the current line. Moreover, trailing "+" and "-" characters have cumulative effect, so '--' refers to the current line less 2.

10. To maintain compatibility with earlier versions of the editor, the character '^' in addresses is equivalent to '-'.

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

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

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

As mentioned, it is generally illegal for more than one command to appear on a line. However, most commands may be suffixed by 'p' or by 'l', in which case the current line is either printed or listed respectively in the way discussed below. Commands may also be suffixed by 'n', meaning the output of the command is to be line numbered. These suffixes may be combined in any order.

(.)a
<text>
.

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

(., .)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. "." is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

(., .)d

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

e filename

The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. "." is set to the last line of the buffer. The number of characters read is typed. "filename" is remembered for possible use as a default file name in a subsequent _r_ or _w_ command. If 'filename' is missing, the remembered name is used.

E filename

This command is the same as _e_, except that no diagnostic results when no _w_ has been given since the last buffer alteration.

f filename

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

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

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

(.)i

<text>
.

This command inserts the given text before the addressed line. "." is left at the last line input, or, if there were none, at the line before the addressed line. This command differs from the _a_ command only in the placement of the text.

(., .+1)j
This command joins the addressed lines into a single line; intermediate newlines simply disappear. "." is left at the resulting line.

(.)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 two-digit octal, and long lines are folded. The _l_ command may be placed on the same line after any non-i/o command.

(., .)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.

(., .)n
The number command prints the addressed lines with line numbers and a tab at the left.

(., .)p
The print command prints the addressed lines. "." is left at the last line printed. The _p_ command may be placed on the same line after any non-i/o command.

(., .)P
This command is a synonym for _p_.

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

Q This command is the same as _q_, except that no diagnostic results when no _w_ has been given since the last buffer alteration.

($)r filename
The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see _e_ and _f_ commands). The file name is remembered if there was no remembered file name already. Address ´0´ is legal for _r_ and causes the file to be read at the beginning of the buffer. If the

read is successful, the number of characters read is typed. "." is left at the last line read in from the file.

(., .)s/regular expression/replacement/ or,
(., .)s/regular expression/replacement/g
The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator 'g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any punctuation character may be used instead of '/' to delimit the regular expression and the replacement. "." is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of '&' in this context may be suppressed by preceding it by '\'. The characters '\n' where n is a digit, are replaced by the text matched by the n-th regular subexpression enclosed between '\(' and '\)'. When nested, parenthesized subexpressions are present, n is determined by counting occurrences of '\(' starting from the left.

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

One or two trailing delimiters may be omitted, implying the 'p' suffix. The special form 's' followed by no delimiters repeats the most recent substitute command on the addressed lines. The 's' may be followed by the letters r (use the most recent regular expression for the left hand side, instead of the most recent left hand side of a substitute command), p (complement the setting of the p suffix from the previous substitution), or g (complement the setting of the g suffix). These letters may be combined in any order.

(., .)ta
This command acts just like the m command, except that a copy of the addressed lines is placed after address a (which may be 0). "." is left on the last line of the copy.

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

(1, $)w filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created. The file name is remembered if there was no remembered file name already. If no file name is given, the remembered file name, if any, is used (see e and

f commands). "." is unchanged. If the command is successful, the number of characters written is printed.

(1, $)W filename
: This command is the same as *w*, except that the addressed lines are appended to the file.

x
: A key string is demanded from the standard input. Later *r*, *e* and *w* commands will encrypt and decrypt the text with this key by the algorithm of *crypt*(1). An explicitly empty key turns off encryption.

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

!<shell command>
: The remainder of the line after the '!' is sent to *sh*(1) to be interpreted as a command. '.' is unchanged.

(.+1,.+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 two addresses are present with no intervening semicolon, *ed* prints the range of lines. If they are separated by a semicolon, the second line is printed.

If an interrupt signal (ASCII DEL) is sent, *ed* prints '?interrupted' 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, on mini computers, 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 2 words.

When reading a file, *ed* discards ASCII NUL characters and all characters after the last newline. It refuses to read files containing non-ASCII characters.

FILES

/tmp/e*
edhup: work is saved here if terminal hangs up
/lib/makekey generate encryption key

SEE ALSO

sed(1)
B. W. Kernighan, *A Tutorial Introduction to the ED Text Editor*
B. W. Kernighan, *Advanced editing on UNIX*

DIAGNOSTICS

"?name" for inaccessible file; "?self-explanatory message" for other errors.

To protect against throwing away valuable work, a _q_ or _e_ command is considered to be in error, unless a _w_ has occurred since the last buffer change. A second _q_ or _e_ will be obeyed regardless.

BUGS

The _l_ command mishandles DEL.
The _undo_ command causes marks to be lost on affected lines.

NAME

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

SYNOPSIS

edit [-r] name ...

DESCRIPTION

Edit is a variant of the text editor _ex_ recommended for new or casual users who wish to use a command oriented editor. The following brief introduction should help you get started with _edit_. A more complete basic introduction is provided by _Edit_: _A tutorial_ . The _Ex_/_edit command summary_ (_version 2.0_) is also very useful. See _ex_(1) for other useful documents; in particular, if you are using a CRT terminal you will want to learn about the display editor _vi_.

BRIEF INTRODUCTION

To edit the contents of an existing file you begin with the command "edit name" to the shell. _Edit_ makes a copy of the file which you can then edit, and tells you how many lines and characters are in the file. To create a new file, just make up a name for the file and try to run _edit_ on it; you will cause an error diagnostic, but don't worry.

Edit prompts for commands with the character ':', which you should see after starting the editor. If you are editing an existing file, then you will have some lines in _edit's_ buffer (its name for the copy of the file you are editing). Most commands to _edit_ use its "current line" if you don't tell them which line to use. Thus if you say print (which can be abbreviated p) and hit carriage return (as you should after all _edit_ commands) this current line will be printed. If you delete (d) the current line, _edit_ will print the new current line. When you start editing, _edit_ makes the last line of the file the current line. If you delete this last line, then the new last line becomes the current one. In general, after a delete, the next line in the file becomes the current line. (Deleting the last line is a special case.)

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

Edit numbers the lines in the buffer, with the first line having number 1. If you give the command "1" then _edit_ will type this first line. If you then give the command delete _edit_ will delete the first line, and line 2 will become line 1, and _edit_ will print the current line (the new line 1) so you can see where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the substitute (s) command. You say "s/_old_/_new_/" where _old_ is replaced by the old characters you want to get rid of and _new_ is the new characters you want to replace it with.

The command file (f) will tell you how many lines there are in the buffer you are editing and will say "[Modified]" if you have changed it. After modifying a file you can put the buffer text back to replace the file by giving a write (w) command. You can then leave the editor by issuing a quit (q) command. If you run _edit_ on a file, but don't change it, it is not necessary (but does no harm) to write the file back. If you try to quit from _edit_ after modifying the buffer without writing it out, you will be warned that there has been "No write since last change" and _edit_ will await another command. If you wish not to write the buffer out then you can issue another quit command. The buffer is then irretrievably discarded, and you return to the shell.

By using the delete and append commands, and giving line numbers to see lines in the file you can make any changes you desire. You should learn at least a few more things, however, if you are to use _edit_ more than a few times.

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

The undo (u) command will reverse the effect of the last command you gave which changed the buffer. Thus if give a substitute command which doesn't do what you want, you can say undo and the old contents of the line will be restored. You can also undo an undo command so that you can continue to change your mind. _Edit_ will give you a warning message when commands you do affect more than one line of the buffer. If the amount of change seems unreasonable, you should consider doing an _undo_ and looking to see what happened. If you decide that the change is ok, then you can _undo_ again to get it back. Note that commands such as _write_ and _quit_ cannot be undone.

To look at the next line in the buffer you can just hit carriage return. To look at a number of lines hit ^D (control key and, while it is held down D key, then let up both) rather than carriage return. This will show you a half screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at the text around where you are by giving the command "z.". The current line will then be the last line printed; you can get back to the line where you were before the "z." command by saying "' The z command can also be given other following characters "z-" prints a screen of text (or 24 lines) ending where you are; "z+" prints the next screenful. If you want less than a screenful of lines do, e.g., "z.12" to get 12 lines total. This method of giving counts works in general; thus you can delete 5 lines starting with the current line with the command "delete 5".

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

The current line has a symbolic name "."; this is most useful in a range of lines as in ".,$print" which prints the rest of the lines in the file. To get to the last line in the file you can refer to it by its symbolic name "$". Thus the command "$ delete" or "$d" deletes the last line in the file, no matter which line was the current line before. Arithmetic with line references is also possible. Thus the line "$-5" is the fifth before the last, and ".+20" is 20 lines after the present.

You can find out which line you are at by doing ".=". This is useful if you wish to move or copy a section of text within a file or between files. Find out the first and last line numbers you wish to copy or move (say 10 to 20). For a move you can then say "10,20move "a" which deletes these lines from the file and places them in a buffer named _a_. _Edit_ has 26 such buffers named _a_ through _z_. You can later get these lines back by doing ""a move ." to put the contents of buffer _a_ after the current line. If you want to move or copy these lines between files you can give an edit (e) command after copying the lines, following it with the name of the other file you wish to edit, i.e. "edit chapter2". By changing _move_ to _copy_ above you can get a pattern for copying lines. If the text you wish to move or copy is all within one file then you can just say "10,20move $" for example. It is not necessary to use named buffers in this case (but you can if you wish).

SEE ALSO

ex(1), vi(1), "Edit: A tutorial", by Ricki Blau and James Joyce

AUTHOR

William Joy

BUGS

See _ex_(_1_).

NAME

egrep - search a file for a pattern

SYNOPSIS

egrep [option] ... [expression] [file] ...

DESCRIPTION

Commands of the _grep_ family search the input _files_ (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. _Egrep_ patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its relative line number in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

-e _expression_
    Same as a simple _expression_ argument, but useful when the _expression_ begins with a -.

-f _file_
    The regular expression is taken from the named _file_ which contains a list of regular expressions to be matched. Each regular expression should appear on a separate line.

The file names are shown in the output if more than one file was searched.

Care should be taken when using the characters $ * [^ | ( ) and \ in the _expression_ as they are also meaningful to the Shell. It is safest to enclose the entire _expression_ argument in single or double quotes.

Egrep accepts regular expressions and it also can accept patterns with "metacharacters". The metacharacter matching protocol is as follows: (note that newline is not considered to be a 'character').

A \ followed by a single character other than newline matches that character.

The character ^ ($) matches the beginning (end) of a line.

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in 'a-z0-9'. A] may occur only as the first character of the string. A literal - must be placed where it can't be mistaken as a range indicator.

A regular expression followed by * (+, ?) matches a sequence of 0 or more (1 or more, 0 or 1) matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

EXAMPLE

```
egrep '^This | match* | regular | expression$' file1 file2 file3
```

will cause all the lines in the three files to be printed that match <u>any</u> of the patterns:

1. a line beginning with 'This'
2. a line containing 'matc' followed by any number of h's
3. a line containing 'regular'
4. a line ending with 'expression'

SEE ALSO

ex(1), fgrep(1), grep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

BUGS

Ideally there should be only one <u>grep</u>, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.

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

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* (∞) and shorthands like >= ($\geqslant$) −> ($\rightarrow$), != ($\neq$), are recognized. Spell out Greek letters in the desired case, as in *alpha, GAMMA.* Mathematical words like sin, cos, log are made Roman automatically. Troff (I) four-character escapes like \(rh ☛) 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.)
NROFF/TROFF User's Manual
troff (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

ex, edit - text editor

SYNOPSIS

ex [-] [-v] [-t tag] [-r] [+*command*] name ...
edit [ex options]

DESCRIPTION

Ex is the root of a family of editors: *edit*, *ex* and *vi*. *Ex* is a superset of *edit*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have not used *ed*, or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed*.

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(1), which is a command which focuses on the display editing portion of *ex*.

The following options are recognized:

- suppresses all interactive-user feedback, as when processing editor scripts in command files.

-v Equivalent to using *vi* rather than *ex*.

-t Equivalent to an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition.

-r Used in recovering after an editor or system crash, retrieving the last saved version of the named file. If no file is specified, a list of saved files will be reported.

+*command*
Indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to $, positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form /pat or line numbers, e.g. +100 to start at line 100.

Name arguments indicate files to be edited.

DOCUMENTATION

The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers or the UNIX system.

The *Ex Reference Manual - Version 3.5/2.13* is a comprehensive and complete manual for the command mode features of *ex*, but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex* see the editing documents written by Brian Kernighan for the editor _ed_; the material in the introductory and advanced documents works also with _ex_.

An Introduction to Display Editing with Vi introduces the display editor _vi_ and provides reference material on _vi_. All of these documents can be found in volume 2c of the Programmer's Manual. In addition, the _Vi Quick Reference_ card summarizes the commands of _vi_ in a useful, functional way, and is useful with the _Introduction_.

FILES

/usr/lib/ex3.6strings	error messages
/usr/lib/ex3.6recover	recover command
/usr/lib/ex3.6preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup command file, user-created in home directory
/tmp/EX_nnnnn_	editor temporary
/tmp/Rx_nnnnn_	named buffer temporary
/usr/preserve	preservation directory
/usr/lib/tags	standard editor tag file

SEE ALSO

awk(1), ed(1), edit(1), grep(1), sed(1), vi(1)

AUTHOR

Originally written by William Joy
Mark Horton has maintained the editor since version 2.7, adding macros, support for many unusual terminals, and other features such as word abbreviation mode.

BUGS

The _undo_ command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The _z_ command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line '-' option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

expr - evaluate arguments as an expression

SYNOPSIS

expr arg ...

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Each token of the expression is a separate argument.

The operators and keywords are listed below. The list is in order of increasing precedence, with equal precedence operators grouped.

expr | *expr*
: yields the first *expr* if it is neither null nor \`0', otherwise yields the second *expr*.

expr & *expr*
: yields the first *expr* if neither *expr* is null or \`0', otherwise yields \`0'.

expr relop expr
: where *relop is one of* < <= = != >= >, yields \`1' if the indicated comparison is true, \`0' if false. The comparison is numeric if both *expr* are integers, otherwise lexicographic.

expr + *expr*
expr - *expr*
: addition or subtraction of the arguments.

expr * *expr*
expr / *expr*
expr % *expr*
: multiplication, division, or remainder of the arguments.

expr : *expr*
: The matching operator compares the string first argument with the regular expression second argument; regular expression syntax is the same as that of *ed*(1). The \(...\) pattern symbols can be used to select a portion of the first argument. Otherwise, the matching operator yields the number of characters matched (\`0' on failure).

(*expr*)
: parentheses for grouping.

EXAMPLES

To add 1 to the Shell variable *a*:

```
a=`expr $a + 1`
```

To find the filename part (least significant part) of the pathname stored in variable _a_, which may or may not contain `/':

```
expr $a : '.*/\(.*\)' '|' $a
```

Note the quoted Shell metacharacters.

SEE ALSO

ed(1), sh(1), test(1)

DIAGNOSTICS

Expr returns the following exit codes:

0	if the expression is neither null nor `0',
1	if the expression is null or `0',
2	for invalid expressions.

NAME

exterr - turn on/off the extended errors in the specified device

SYNOPSIS

exterr /dev/devicename [yn]

DESCRIPTION

Exterr turns on [or off] the reporting of extended errors on the specified device.

If reporting of errors is turned "off" with the argument _n_, only fatal errors are reported.

The default condition is "yes", in which case soft as well as hard errors are reported on the specified device. The devicename must be the "raw" one to access the ioctl.

NAME

f77 - FORTRAN compiler

SYNOPSIS

f77 [-o ofile] [-i] [-c] [-u] [-v] file ...

DESCRIPTION

f77, the FORTRAN compiler, accepts a list of FORTRAN source files and various intermediate texts contained in the list of files specified by _file_ and puts the resulting executable object module in _a.out_ (but see the -o option, described below).

In order to understand the use of f77, the reader must first understand the steps which the compiler goes through in order to turn a FORTRAN source program into an executable object file.

The FORTRAN compiler generates several intermediate files on the way to generating the final executable file. The first phase of the compiler generates an intermediate file, of the same name as the source file, but with a .i suffix. This intermediate file is destined for processing by the code generator.

The code generator is the second phase of the process. The output of the code generator is a file with the same name as the source file, but with a suffix of .obj. The *.obj file is the input to the next phase, called ulinker.

The ulinker phase of the compilation process converts the .obj file into a UNIX-style object file with a .o suffix. This file can then be processed by the UNIX loader utility, ld.

Finally, the ld utility produces the final executable code file.

When using f77, any combination of FORTRAN source files (each having a .for suffix) can be combined with FORTRAN or Pascal intermediate files (each having a .i suffix), FORTRAN or Pascal object code files (each having a .obj suffix), and UNIX object files (each having a .o suffix). When the compilation completes successfully, the result of the combination of all those files is placed in the file _a.out_ or in the file specified by the -o option.

The -o option, if given, specifies that the file _ofile_ (runnable file) whose name follows the option is the file to receive the final executable code. If the -o option is not specified, the resultant executable file is placed in the file _a.out_.

If the -i option is given, the FORTRAN intermediate code (the result of running /_lib_/_fortran_) is placed in a file of the same name as the source file, but with a suffix of .i appended. The compilation then terminates.

If the -c option is given, the FORTRAN unlinked object code (the result of running /*lib*/*code*) is placed in a file of the same name as the source file, but with a suffix of .obj appended. The compilation then terminates.

If the -u option is given, the linked object code (the result of running /*lib*/*ulinker*) is placed in a file of the same name as the source file, but with a suffix of .o appended. The compilation then terminates.

The -v (for verbose) option makes f77 display a running progress report as it compiles.

If only one *file* argument is supplied on the command line, then all the intermediate files (.i, .obj, .o) are removed at the end of the compilation. If multiple *file* arguments are typed on the command line, any existing intermediate files are not removed.

EXAMPLES

```
f77 progl.for
```

compiles *progl*.*for* and puts the resulting object module in *a*.*out*.

```
f77 -o frammis prog2.for prog3.obj
```

compiles the FORTRAN program called *prog2*.*for* and links the result with the object file *prog3*.*obj*. The result of the compilation is placed in the output file called *frammis*.

FILES

*.for	FORTRAN source
*.i	Intermediate code
*.obj	Compiled unlinked f77 object
*.o	Compiled unlinked UNIX object
/lib/ftnlib.obj	
/lib/paslib.obj	
/lib/fortran	
/lib/code	
/lib/ulinker	
/lib/ftncterrs	
/lib/ftnrterrs	
/bin/ld	linking loader
/lib/crt0.o	startup routine

SEE ALSO

"User Documentation Update for UniSoft Pascal and FORTRAN".

NAME

true, false - provide truth values

SYNOPSIS

true

false

DESCRIPTION

True and _false_ are usually used in a Bourne shell script. They return the appropriate status "true" or "false".

EXAMPLE

```
while false
do
     command list
done
```

SEE ALSO

csh(1), sh(1), true(1)

DIAGNOSTICS

False has exit status nonzero.

NAME

fgrep - search a file for a pattern

SYNOPSIS

fgrep [option] ... [strings] [file]

DESCRIPTION

Commands of the _grep_ family search the input _files_ (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. _Fgrep_ patterns are fixed strings; it is fast and compact. The following options are recognized.

-v All lines but those matching are printed.

-x (Exact) only lines matched in their entirety are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its relative line number in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

-e _expression_
Same as a simple _expression_ argument, but useful when the _expression_ begins with a -.

-f _file_
The string list (_fgrep_) is taken from the _file_.

In all cases the file name is shown if there is more than one input file. Care should be taken when using the characters $ * [^ | ( ) and \ in the _expression_ as they are also meaningful to the Shell. It is safest to enclose the entire _expression_ argument in single quotes ´ ´.

Fgrep searches for lines that contain one of the (newline-separated) _strings_.

Regular expressions given to _fgrep_ must be enclosed in single quotes and a backslash (\) must immediately precede the newline between strings. The newline or carriage return itself is not considered to be a character. _Fgrep_ searches only for fixed strings that match exactly and will not accept metacharacter matching, as will _egrep_ (q.v.).

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

EXAMPLE

```
fgrep -n ' string1\
string2\
string3\ ' file1 file2 file3
```

reports the lines and line numbers from each of the three files that contain the specified strings. Note that the string list is enclosed in both single quotes and blanks. Do not put a space between the backslash and the newline (carriage return).

SEE ALSO

egrep(1), ex(1), grep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

BUGS

Ideally there should be only one _grep_, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.

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

NAME

file - determine file type

SYNOPSIS

file file ...

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify the file(s) by type. If an argument appears to be ascii, _file_ examines the first 512 bytes and tries to guess its language.

EXAMPLE

file textfile programfile directory

reports the file names and directory name, and whether the files are English text, nroff input, a C program, or whatever.

DIAGNOSTICS

If _file_ cannot decipher a filetype, it reports "cannot stat".

BUGS

It often makes mistakes. In particular it often suggests that command files are C programs.

NAME
find - find files

SYNOPSIS
find pathname-list predicate-list expression

DESCRIPTION
Find recursively descends the directory hierarchy one directory at a time, for each pathname in the *pathname-list* (i.e., one or more pathnames) using the first pathname in the list as the starting point.

You can use *find* to locate files for which you can remember the name but not the location, or to locate files that fulfill certain criteria.

Find seeks files that match conditions set forth in the *predicate-list*, and performs actions specified in the *expression*.

In the predicate-list, the number argument *n* is used to mean a decimal integer where +*n* means more than *n*, -*n* means less than *n* and *n* means exactly *n*.

The following predicate descriptors are available:

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

-perm onum
: True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(2)) become significant and the flags are compared: (*flags*&*onum*)==*onum*.

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

-links n
: True if the file has *n* links.

-user uname
: True if the file belongs to the user *uname* (login name or numeric user ID).

-group gname
: True if the file belongs to group *gname* (group name or numeric group ID).

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

-inum n
: True if the file has inode number *n*.

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

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

-exec command
True if the executed command returns a zero value as exit status. The end of the -exec and command sequence must consist of a pair of curly braces and an escaped semicolon. With -exec the command argument '{}' is necessary to store the current pathname.

-ok command
Like -exec in its syntax, except that the generated command is written on the standard output, then the standard input is read and the command executed only upon response "yes", or y.

-print Always true; causes the current pathname to be printed. Do not terminate this command with curly braces or a semicolon.

-newer file
True if the current file has been modified more recently than the argument file.

The primaries or predicate operators may be combined using the following operators (in order of decreasing precedence):

1) A parenthesized group of primaries and operators (parentheses are special to the Shell and must be escaped).

2) The negation of a primary ('!' is the unary not operator).

3) Concatenation of primaries (the and operation is implied by the juxtaposition of two primaries).

4) Alternation of primaries ('-o' is the or operator).

EXAMPLES

```
find / -perm 755 -exec ls "{}" ";"
```

will find all files, starting with the root directory, on which the permission levels have been set to 755 (see chmod(1)).

With -exec and a command such as ls, it is often necessary to escape the "{}" that stores the current pathname under investigation by putting it in double quotes. It is always necessary to escape the semicolon at the end of an -exec sequence.

Note again that it is also necessary to escape parentheses " \(" and " \) " used for grouping primaries, by means of a backslash.

FILES

/etc/passwd
/etc/group

SEE ALSO

sh(1)

BUGS

The syntax is painful.

NAME

freq - report on character frequencies in a file

SYNOPSIS

freq [file ...]

DESCRIPTION

Freq counts occurrences of characters in the list of files specified on the command line. If no files are specified, the standard input is read.

EXAMPLE

The example below shows *freq* used to count characters in the source text for this manual page:

```
freq /usr/man/man1/freq.1
|nul     0|soh     0|stx     0|etx     0|
|eot     0|enq     0|ack     0|bel     0|
|bs      0|ht      0|lf     61|vt      0|
|ff      0|cr      0|so      0|si      0|
|dle     0|dc1     0|dc2     0|dc3     0|
|dc4     0|nak     0|syn     0|etb     0|
|can     0|em      0|sub     0|esc     0|
|fs      0|gs      0|rs      0|us      0|
|      193|!       0|"       2|#       0|
|$       0|%       0|&       0|'       0|
|(       3|)       3|*       0|+       2|
|,       4|-      13|.      39|/       0|
|0       0|1       4|2       0|3       0|
|4       0|5       0|6       0|7       0|
|8       2|9       0|:       1|;       2|
|<       0|=       0|>       0|?       0|
|@       0|A       3|B      13|C       1|
|D       1|E       5|F       2|G       0|
|H       5|I      12|J       0|K       0|
|L       1|M       1|N       4|O       3|
|P      10|Q       1|R       7|S      10|
|T      10|U       1|V       0|W       0|
|X       0|Y       1|Z       0|[       5|
|\      11|]       5|^       0|_       0|
|`       0|a      60|b      13|c      33|
|d      39|e     125|f      29|g      12|
|h      33|i      62|j       4|k       3|
|l      23|m      15|n      69|o      57|
|p      31|q       4|r      59|s      54|
|t      80|u      32|v       1|w       4|
|x       0|y       6|z       0|{       0|
||       0|}       0|~       0|del     0|
```

NAME

fsck - file system consistency check and interactive repair

SYNOPSIS

fsck [-y] [-n] [-sX] [-SX] [-t filename] [filesystem] ...

DESCRIPTION

fsck audits and interactively repairs inconsistent conditions for file systems. If the file system is inconsistent the operator is prompted for concurrence before each correction is attempted. It should be noted that a number of the corrective actions will result in some loss of data. The amount and severity of data lost may be determined from the diagnostic output. The default action for each consistency correction is to wait for the operator to respond yes or no. If the operator does not have write permission _fsck_ will default to a -n action.

Fsck has more consistency checks than its predecessors _check_, _dcheck_, _fcheck_, and _icheck_ combined.

The following flags are interpreted by _fsck_.

-y Assume a yes response to all questions asked by _fsck_; this should be used with great caution as this is a free license to continue after essentially unlimited trouble has been encountered.

-n Assume a no response to all questions asked by _fsck_; do not open the file system for writing.

-s_X_ Ignore the actual free list and (unconditionally) reconstruct a new one by rewriting the super-block of the file system. The file system should be unmounted while this is done; if this is not possible, care should be taken that the system is quiescent and that it is rebooted immediately afterwards. This precaution is necessary so that the old, bad, in-core copy of the superblock will not continue to be used, or written on the file system.

The -s_X_ option allows for creating an optimal free-list organization. The following forms of _X_ are supported for the following devices:

-s3 (RP03)
-s4 (RP04, RP05, RP06)
-sBlocks-per-cylinder:Blocks-to-skip (for anything else)

If _X_ is not given, the values used when the filesystem was created are used. If these values were not specified, then the value _400_:_9_ is used.

-S_X_ Conditionally reconstruct the free list. This option is like -s_X_ above except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using -S will force a no response to all questions asked by _fsck_. This option is useful for forcing free list reorganization on uncontaminated file systems.

-t If *fsck* cannot obtain enough memory to keep its tables, it uses a scratch file. If the -t option is specified, the file named in the next argument is used as the scratch file, if needed. Without the -t flag, *fsck* will prompt the operator for the name of the scratch file. The file chosen should not be on the filesystem being checked, and if it is not a special file or did not already exist, it is removed when *fsck* completes.

If no filesystems are given to *fsck* then a default list of file systems is read from the file /etc/checklist.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one inode or the free list.
2. Blocks claimed by an inode or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Directory size not 16-byte aligned.
5. Bad inode format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated inode.
   Inode number out of range.
8. Super Block checks:
   More than 65536 inodes.
   More blocks for inodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the lost+found directory. The name assigned is the inode number. The only restriction is that the directory lost+found must preexist in the root of the filesystem being checked and must have empty slots in which entries can be made. This is accomplished by making lost+found, copying a number of files to the directory, and then removing them (before *fsck* is executed).

Checking the raw device is almost always faster.

EXAMPLE

```
fsck /dev/rdisk0
```

checks the consistency of device *rdisk0*.

FILES

/etc/checklist   contains default list of file systems to check.

DIAGNOSTICS

The diagnostics produced by *fsck* are intended to be self-explanatory.

SEE ALSO

dcheck(1M), icheck(1M)

BUGS

Inode numbers for . and .. in each directory should be checked for validity.

-g and -b options from *check* should be available in *fsck*.

NAME

get - get a version of an SCCS file

SYNOPSIS

get [-rSID] [-ccutoff] [-ilist] [-xlist] [-aseq-no.] [-k] [-e] [-l[p]] [-p] [-m] [-n] [-s] [-b] [-g] [-t] file ...

DESCRIPTION

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

The generated text is normally written into a file called the _g-file_ whose name is derived from the SCCS file name by simply removing the leading s.; (see also _FILES_, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

-r_SID_ The _S_CCS _ID_entification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by _delta_(1) if the -e keyletter is also used), as a function of the SID specified.

-c_cutoff_ _Cutoff_ date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified _cutoff_ date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, -c7502 is equivalent to -c750228235959. Any number of non-numeric characters may separate the various 2 digit pieces of the _cutoff_ date-time. This feature allows one to specify a _cutoff_ date in the form: "-c77/2/2 9:22:25". Note that this implies that one may use the %E% and %U% identification keywords (see below) for nested _gets_ within, say the input to a _send_(1C) command:

~!get "-c%E% %U%" s.file

-e Indicates that the _get_ is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(1). The -e keyletter used in a *get* for a particular version (SID) of the SCCS file prevents further *gets* for editing on the same SID until *delta* is executed or the j (joint edit) flag is set in the SCCS file (see *admin*(1)). Concurrent use of get -e for different SIDs is always allowed.

If the *g-file* generated by *get* with an -e keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the *get* command with the -k keyletter in place of the -e keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see *admin*(1)) are enforced when the -e keyletter is used.

-b — Used with the -e keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the b flag is not present in the file (see *admin*(1)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)
Note: A branch *delta* may always be created from a non-leaf *delta*.

-i*list* — A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

```
<list> ::= <range> | <list> , <range>
<range> ::= SID | SID - SID
```

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

-x*list* — A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the -i keyletter for the *list* format.

-k — Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The -k keyletter is implied by the -e keyletter.

-l[p] — Causes a delta summary to be written into an *l-file*. If -lp is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

-p — Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output which normally goes to the standard output goes to file descriptor 2 instead, unless the -s keyletter is used, in which case it disappears.

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

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

-n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the -m and -n keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the -m keyletter generated format.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

-t Used to access the most recently created ("top") delta in a given release (e.g., -r1), or release and level (e.g., -r1.2).

-a*seq-no*. The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(5)). This keyletter is used by the *comb*(1) command; it is not a generally useful keyletter, and users should not use it. If both the -r and -a keyletters are specified, the -a keyletter is used. Care should be taken when using the -a keyletter in conjunction with the -e keyletter, as the SID of the delta to be created may not be what one expects. The -r keyletter can be used with the -a and -e keyletters to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the -e keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the -i keyletter is used included deltas are listed following the notation "Included"; if the -x keyletter is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	-b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	-	R < mR and R does <u>not</u> exist	hR.mL**	hR.mL.(mB+1).1
R	-	Trunk succ.# in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	-	Trunk succ. in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	-	Branch succ.	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the <u>new</u> branch (that is, maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components <u>must</u> exist.

** "hR" is the highest _existing_ release that is lower than the specified, _nonexistent_, release R.

*** This is used to force creation of the _first_ delta in a _new_ release.

Successor.

† The -b keyletter is effective only if the b flag (see _admin_(1)) is present in the file. An entry of - means "irrelevant".

‡ This case applies if the d (default SID) flag is _not_ present in the file. If the d flag _is_ present in the file, then the SID obtained from the d flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing _identification keywords_ with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	Value
%M%	Module name: either the value of the m flag in the file (see _admin_(1)), or if absent, the name of the SCCS file with the leading s. removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).
%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the t flag in the SCCS file (see _admin_(1)).
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the q flag in the file (see _admin_(1)).
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this shouldn't have happened" type errors. It is _not_ intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by _what_(1).
%W%	A shorthand notation for constructing _what_(1) strings for UNIX program files. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing _what_(1) strings for non-UNIX program files. %A% = %Z%%Y% %M% %I%%Z%

FILES

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

The *g-file*, which contains the generated text, is created in the current directory (unless the -p keyletter is used). A *g-file* is created in all cases, whether or not any lines of text were generated by the *get*. It is owned by the real user. If the -k keyletter is used or implied its mode is 644; otherwise its mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the -l keyletter is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

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

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

The *p-file* is used to pass information resulting from a *get* with an -e keyletter along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an -e keyletter for the same SID until *delta* is executed or the joint edit flag, j, (see *admin*(1)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the -i keyletter argument if it was present, followed by a blank and the -x keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

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

SEE ALSO

admin(1), delta(1), help(1), prs(1), what(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

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

NAME

getty - set terminal mode

SYNOPSIS

/etc/getty [char]

DESCRIPTION

Getty is invoked by *init*(1M) immediately after a terminal is opened, following the making of a connection. 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 terminal line. (see ttys(5)). Normally, it sets the speed of the interface, specifies that raw mode is to be used (break on every character), that echo is to be suppressed, and either parity allowed. It types a banner identifying the system (from /etc/ident) and the 'login:' message. Then the user's name is read, a character at a time.

If a null or break character is received and the parameter to *getty* specifies a multiple speed line, *getty* will step to the next baud rate and start again.

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*(2)).

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(1M), login(1), stty(2), ttys(5)

NAME

grep - search a file for a pattern

SYNOPSIS

grep [option] ... expression [file] ...

DESCRIPTION

Commands of the *grep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Grep* patterns are limited regular expressions in the style of *ex*(*1*); it uses a compact nondeterministic algorithm.

The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its relative line number in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

-e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a -.

In all cases the file name is shown if there is more than one input file. Care should be taken when using the characters $ * [^ | ( ) and \ in the *expression* as they are also meaningful to the Shell. It is safest to enclose the entire *expression* argument in single quotes ′ ′.

Grep accepts metacharacter matching characters as well as fixed regular expressions. The metacharacter matching protocol is as follows: (note that newline is not considered to be a ′character′).

A \ followed by a single character other than newline matches that character.

The character ^ ($) matches the beginning (end) of a line.

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in 'a-z0-9'. A] may occur only as the first character of the string. A literal - must be placed where it can't be mistaken as a range indicator.

A regular expression followed by * (+, ?) matches a sequence of 0 or more (1 or more, 0 or 1) matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

EXAMPLE

```
grep -v -c 'regular' grep.1
```

reports a count of the number of lines that do _not_ contain the word _regular_ in the file _grep.1_.

SEE ALSO

egrep(1), ex(1), fgrep(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

BUGS

Ideally there should be only one _grep_, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.

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

NAME

head - give first few lines

SYNOPSIS

head [-count] [file ...]

DESCRIPTION

This filter gives the first count lines of each of the specified files, or of the standard input. If count is omitted it defaults to 10.

EXAMPLE

```
head -6 filea fileb filec
```

will print out the first six lines of the three specified files. The filename will appear before each new set of head lines listed, if more than one file has been specified.

SEE ALSO

tail(1)

NAME

help - ask for help about SCCS problems.

SYNOPSIS

help [args]

DESCRIPTION

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

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

type 1 Begins with non-numerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., ge6, for message 6 from the *get* command).

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

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

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

When all else fails, try "help stuck".

FILES

/usr/lib/help directory containing files of message text.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

hex - translates object files into ASCII formats suitable for Motorola S-record downloading.

SYNOPSIS

hex [-1] [-n#] [-s0] [-s2] [-ns8] [+saddr] ifile

DESCRIPTION

hex translates object files into ASCII formats suitable for Motorola S-record downloading. The following options determine locations:

- **1** Output 'Loading at' message.
- **n#** Number of characters to output per record. # is a decimal number.
- **s0** Output a leading s0 record.
- **s2** S2 records only (no s1 records are produced).
- **ns8** Do not output a trailing s8 (s9) record.
- **saddr** Starting load address (in hex).
- **ifile** File to be downloaded. The file's starting address is used if saddr is not present.

AUTHOR

Jeff Schriebman, August 1981

NAME

icheck - file system storage consistency check

SYNOPSIS

icheck [-s] [-b numbers] [filesystem]

DESCRIPTION

N.B.: _Icheck_ has been made obsolete for normal consistency checking by _fsck_(1M).

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

> The total number of files and the numbers of regular, directory, block special and character special files.
>
> The total number of blocks in use and the numbers of single-, double-, and triple-indirect blocks and directory blocks.
>
> The number of free blocks.
>
> The number of blocks missing; i.e. not in any file nor in the free list.

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

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

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

EXAMPLE

```
icheck /dev/rdisk0
```

checks the consistency of the file system storage on device _rdisk0_.

FILES

/etc/checklist

SEE ALSO

clri(1M), dcheck(1M), fsck(1M), ncheck(1M)

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.

The system should be fixed so that the reboot after fixing the root file system is not necessary.

NAME

init - process control initialization

SYNOPSIS

init

DESCRIPTION

Init is invoked inside the system as the last step in the boot procedure. _Init_ commences single user operation by giving the super-user a shell on the console.

When such single user operation is terminated by killing the single-user shell (i.e. by hitting Control-d), _init_ runs /_etc_/_rc_. This command file performs housekeeping operations such as removing temporary files, mounting file systems, starting daemons, and the /_etc_/_update_ process.

In multi-user operation, _init's_ role is to create a process for each terminal port on which a user may log in. To begin such operations, it reads the file /_etc_/_ttys_ and forks several times to create a process for each terminal specified in the file. Each of these processes opens the appropriate terminal for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input and output and the diagnostic output.

If a terminal exists but an error occurs when trying to open the terminal _init_ complains by writing a message to the system console. After an open succeeds, /_etc_/_getty_ is called with argument as specified by the second character of the _ttys_ file line. _Getty_ reads the user's name and invokes _login_ to log in the user and execute the Shell. Usually, users will begin by running the C shell, but this can be changed by editing the password file (see _passwd_(_5_)).

Ultimately the Shell will terminate because of an end-of-file (Control-d) 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 terminal is reopened and _getty_ is reinvoked.

Init catches the _hangup_ signal (signal SIGHUP) and interprets it to mean that the file /_etc_/_ttys_ should be read again. The Shell process on each line which used to be active in _ttys_ but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add lines without rebooting the system by changing the _ttys_ file and sending a _hangup_ signal to the _init_ process: use 'kill -1 1' or 'kill -HUP 1.'

Init will terminate multi-user operations and resume single-user mode if sent a terminate (TERM) signal, i.e. "kill -TERM 1". If there are processes outstanding which are deadlocked (due to hardware or software failure), _init_ will not wait for them all to die (which might take forever), but will time out after 30 seconds and print a warning message.

If, at bootstrap time, the *init* process cannot be located, the system will loop in user mode.

DIAGNOSTICS

"init: *tty*: cannot open." A terminal which is turned on in the *rc* file cannot be opened, likely because the requisite lines are either not configured into the system or the associated device was not attached during boot-time system configuration.

WARNING: Something is hung (won't die); `ps axl' advised. A process is hung and could not be killed when the system was shutting down. This is usually caused by a process which is stuck in a device driver due to a persistent device error condition.

FILES

/dev/console	
/dev/tty?	
/etc/utmp	
/usr/adm/wtmp	
/etc/ttys	
/etc/rc	
/etc/update	periodically update the super block
/etc/getty	to set up terminal

SEE ALSO

login(1), kill(1), sh(1), ttys(5), getty(1M)

NAME

join - relational database operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is '-', the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined. If not otherwise stated, *join* normally joins on the first field in each line.

There is one line in the output for each line in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*, but the order of output of the fields can be changed with the -o option, described below.

Fields are normally separated by blank, tab or newline. In this case, multiple separators count as one, and leading separators are discarded. The field separators can be changed if desired.

These options are recognized:

-a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

-e *s* Replace empty output fields by string *s*.

-j*n m* Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.

-o *list*
Each output line comprises the fields specifed in *list*, each element of which has the form *n*.*m*, where *n* is a file number and *m* is a field number.

-t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

EXAMPLE

Consider that we have two files called *people* and *work*, which contain a list of peoples' name and their workplaces. The *people* file contains:

```
Austen
Bailey
Clark
Daniels
Davidson
Dawson
Morgan
Parker
Smith
Williams
```

and the _work_ file contains:

```
Jack Austen Anchor Brewery
Maryann Clark Shoeshop
Steve Daniels McGuiness Distillery
Sylvia Dawson Laphroaig
Henry Morgan Downtown Theatre
Sally Smith Talcum Powdery
Bill Williams Computer Software
```

The example below shows the effects of the _join_ program:

```
join -j1 1  -j2 2  -o 2.1 2.2 2.3 2.4 2.5 people work
Jack Austen Anchor Brewery
Maryann Clark Shoeshop
Steve Daniels McGuiness Distillery
Sylvia Dawson Laphroaig
Henry Morgan Downtown Theatre
Sally Smith Talcum Powdery
Bill Williams Computer Software
```

The join was done between the first field of the _people_ file, and the second field of the _work_ file. The -o option is used to get the output lines in first name - last name order.

For every first field in _people_ which matches the second field in _work_, the output consists of the peoples names followed by their place of work. If the -o option was not used, the peoples' names would appear last name first.

SEE ALSO
sort(1), comm(1), awk(1)

BUGS
With default field separation, the collating sequence is that of _sort_ -_b_; with -t, the sequence is that of a plain sort.

The conventions of _join_, _sort_, _comm_, _uniq_, _look_ and _awk_(1) are wildly incongruous.

NAME

kill - terminate a process with extreme prejudice

SYNOPSIS

kill [-sig] processid ...

DESCRIPTION

Kill sends the TERM (terminate, 15) signal to the specified processes. If a signal name or number preceded by '-' is given as first argument, that signal is sent instead of terminate (see also *signal*(2)). The list of signal names and numbers is stored in */usr/include/signal.h*. Signals are often referred to by their names, stripped of the common SIG prefix.

Here is a list of the signal names and numbers. Signal numbers are not often used directly. The most common usage of the *kill* command is simply ''kill'' plus the process ID number (see *ps*(*1*).

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

N.*B*.: The starred (*) signals generate a core image if not caught or ignored.

The terminate signal will kill processes that do not catch the signal; "kill -9 ..." is a sure kill, as the KILL (9) signal cannot be caught. By convention, if process number 0 is specified, all members in the process group (i.e. processes resulting from the current login) are signaled (but beware: this works only if you use *sh*(1); not if you use *csh*(1).) In order to be killed, a process must belong to you unless you are the super-user.

The process number of an asynchronous process started with '&' is reported by the shell. Process numbers can also be found by using *kill* as a built-in to *csh*(1); See *csh*(1) for details.

EXAMPLE

```
kill 24068
```

stops the process with the I.D. number 24068.

SEE ALSO

csh(1), ps(1), kill(2), signal(2)

BUGS

An option to kill process groups ala _killpg_(2) should be provided; a replacement for ''kill 0'' for _csh_(1) users.

NAME

last - indicate last logins of users and teletypes

SYNOPSIS

last [name ...] [tty ...]

DESCRIPTION

Last will look back in the _wtmp_ file which records all logins and logouts for information about a user, a teletype [terminal] or any group of users and teletypes. Arguments specify names of users or teletypes of interest. Names of teletypes may be given fully or abbreviated. For example 'last 0' is the same as 'last tty0'. If multiple arguments are given, the information which applies to any of the arguments is printed. For example 'last root console' would list all of "root's" sessions as well as all sessions on the console terminal.

Last reports the sessions of the specified users and teletypes, most recent first, indicating start times, duration, and teletype for each. If the session is still continuing or was cut short by a reboot, _last_ so indicates.

EXAMPLE

```
last reboot
```

will give an indication of mean time between reboots of the system.

Last with no arguments prints a record of all logins and logouts, in reverse order. Since _last_ can generate a great deal of output, piping it through the _more_ program for screen viewing is advised.

If _last_ is interrupted with a "break", it indicates how far the search has progressed in _wtmp_. If interrupted with a quit signal (generated by a control-\) _last_ exits and dumps core.

Control-d (EOF) signal does nothing. Therefore exit gracefully from _last_ with a "break" or "shift/delete" signal.

FILES

/usr/adm/wtmp login data base

SEE ALSO

wtmp(5)

AUTHOR

Howard Katseff

NAME

ld - loader

SYNOPSIS

ld [option] file ...

DESCRIPTION

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

The argument routines are concatenated in the order specified. The entry point of the output is the 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 may be important.

The symbols `_etext', `_edata' and `_end' (`etext', `edata' and `end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. It is erroneous to define these symbols.

Ld understands several options. Except for -l, they should appear before the file names.

-s `Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by *strip*(1).

-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*x* This option is an abbreviation for the library name `/lib/lib*x*.a', where *x* is a string. If that does not exist, *ld* tries `/usr/lib/lib*x*.a'. A library is searched when its name is encountered, so the placement of a -l is significant.

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

-X Save local symbols except for those whose names begin with `L'. This option is used by _cc_(1) 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.

-R _x_ Set starting relocation address of program to _x_ (x is in hex).

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

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

-N _x_ Set the data relocation boundary to _x_ for shared text programs. The value _x_ may be followed by a _k_ or _K_ to indicate multiplication by 1024.

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

-e The following argument is taken to be the name of the entry point of the loaded program; location 0 is the default.

-F _x_ Add offset _x_ to all data references (x is in hex).

EXAMPLE

```
ld -s /lib/crt0.o filea.o fileb.o -lc
```

will load subroutines filea with fileb for execution and remove its symbol table.

FILES

/lib/lib*.a	libraries
/usr/lib/lib*.a	more libraries
a.out	default output file
/lib/crt0.o	"C" start up routine

SEE ALSO

as(1), ar(1), cc(1)

NAME

lex - generator of lexical analysis programs

SYNOPSIS

lex [-tvfn] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analyis of text. The input _files_ (standard input default) contain regular expressions to be searched for, and actions written in C to be executed when expressions are found.

A C source program, 'lex.yy.c' is generated, to be compiled thus:

```
cc lex.yy.c -ll
```

This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The options have the following meanings.

-t Place the result on the standard output instead of in file "lex.yy.c".

-v Print a one-line summary of statistics of the generated analyzer.

-n Opposite of -v; -n is default.

-f "Faster" compilation: don't bother to pack the resulting tables; limited to small programs.

EXAMPLE

```
lex lexcommands
```

would draw _lex_ instructions from the file _lexcommands_, and place the output in _lex.yy.c_

```
%%
[A-Z] putchar(yytext[0]+'a'-'A');
[ ]+$
[ ]+  putchar(' ');
```

is an example of a _lex_ program that would be put into a _lex_ command file. This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

FILES

/usr/lib/lex/ncform lex "C" interface

SEE ALSO

yacc(1), sed(1)

M. E. Lesk and E. Schmidt, LEX - Lexical Analyzer Generator

NAME

lint - a C program verifier

SYNOPSIS

lint [-abchnpuvx] file ...

DESCRIPTION

Lint attempts to detect features of the C program *files* which are likely to be bugs, or non-portable, or wasteful. It also checks the type usage of the program more strictly than the compilers.

Among the things which are currently found are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

By default, it is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. Function definitions for certain libraries are available to *lint*; these libraries are referred to by a conventional name, such as \`-lm', in the style of *ld*(1).

Any number of the options in the following list may be used. The -D, -U, and -I options of *cc*(1) are also recognized as separate arguments.

p Attempt to check portability to the *IBM* and *GCOS* dialects of C.

h Apply a number of heuristic tests to attempt to intuit bugs, improve style, and reduce waste.

b Report *break* statements that cannot be reached. (This is not the default because, unfortunately, most *lex* and many *yacc* outputs produce dozens of such comments.)

v Suppress complaints about unused arguments in functions.

x Report variables referred to by extern declarations, but never used.

a Report assignments of long values to int variables.

c Complain about casts which have questionable portability.

u Do not complain about functions and variables used and not defined, or defined and not used (this is suitable for running *lint* on a subset of files out of a larger program).

n Do not check compatibility against the standard library.

Exit(2) and other functions which do not return are not understood; this causes various lies.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
: at appropriate points stops comments about unreachable code.

/*VARARGS*n**/
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*NOSTRICT*/
: shuts off strict type checking in the next expression.

/*ARGSUSED*/
: turns on the -v option for the next function.

/*LINTLIBRARY*/
: at the beginning of a file shuts off complaints about unused functions in this file.

EXAMPLE

The following *lint* call:

```
lint -b myfile.c
```

checks the consistency of the file `myfile.c'. The -b option indicates that unreachable break statements are not to be checked. This option might well be used on files that *lex*(1) generates.

FILES

/lib/lint[12] programs
/lib/llib-lc declarations for standard functions
/lib/llib-port declarations for portable functions

SEE ALSO

cc(1)
S. C. Johnson, *Lint, a C Program Checker*

BUGS

There are some things you just can't get lint to shut up about.

NAME

ln - make links

SYNOPSIS

```
ln name1 [ name2 ]
ln name ... directory
```

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. You can use _link_ to put a file in several directories; or to put a file in another directory under another name. A link is not a copy. Any changes made to the file in one directory will be seen whenever that file is accessed through one of its other links. 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.

Given one or two arguments, _ln_ creates a link to an existing file _name1_. If _name2_ is given, the link has that name; _name2_ may also be a directory in which to place the link; otherwise it is placed in the current directory. If only the directory is specified, the link will be made with its name the same as the last component of _name1_.

Given more than two arguments, _ln_ makes links to all the named files in the named directory. The links made will have the same name as the files being linked to.

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

EXAMPLE

```
ln filea /unisoft/fileb
```

links filea to the name "fileb" in the /unisoft directory.

```
ln filea fileb filec /unisoft
```

will link filea to /unisoft/filea, fileb to /unisoft/fileb, and filec to /unisoft/filec.

SEE ALSO

rm(1), cp(1), mv(1)

NAME
login - sign on

SYNOPSIS
login [username]

DESCRIPTION
The _login_ command is used when a user initially signs on, or it may be used at any time to change from one user to another. The login script begins to run when a Control-d is given to the single-user (#) prompt after booting the system. For further details on initial login, see "How to Get Started" in the Introduction to this volume.

If _login_ is invoked without an argument, it responds with the
login:
prompt, and it expects a valid user name, and, if appropriate, a password. It will not ask for a password unless passwords exists for the user.

Echoing is turned off during the typing of the password, so that the password will remain secure.

After a successful login, accounting files are updated, the user is informed of the existence of mail, and the message of the day (motd) and the time of last login are printed.

Login initializes the user and group IDs and the working directory, then executes a command interpreter (default is _sh_(1)) according to specifications found in a password file. Argument 0 of the command interpreter is -sh, the name of the command interpreter with a leading dash (-) attached.

Login also initializes the environment _environ_(5) with information specifying home directory, command interpreter, terminal type (if available) and user name.

Login is recognized by _sh_(1) and _csh_(1) and executed directly (without forks).

EXAMPLE
login

causes the system to give the prompt,
login:
to which a user name is the appropriate response.

FILES

/etc/utmp	accounting
/usr/adm/wtmp	accounting
/usr/spool/mail/*	mail
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/ttys	terminal initialization data
/etc/ttytype	data base of terminal type by port

SEE ALSO

environ(5), getty(1M), init(1M), mail(1), passwd(1), passwd(5),

DIAGNOSTICS

Login incorrect, if the name or the password is bad.
No Shell, if the shell specified for that user cannot be executed.
No Directory, if the home directory specified for that user does not exist or is protected.

NAME

look - find lines in a sorted list

SYNOPSIS

look [-df] string [file]

DESCRIPTION

Look consults a sorted *file* and prints all lines that begin with *string*. The shell is usually happier if you put double quotation marks around *string*.

The options d and f affect comparisons as in *sort*(1):

d "Dictionary" order: only letters, digits, tabs and blanks participate in comparisons.

f Fold. Upper case letters compare equal to lower case.

If no *file* is specified, /*usr*/*dict*/*words* is assumed with collating sequence -df. You can use this to discover whether a given word is included in the on-line dictionary.

EXAMPLE

```
look -f "This" filea
```

prints all the lines that begin with the word "This", in upper or lower case.

FILES

/usr/dict/words

SEE ALSO

sort(1), grep(1)

NAME

lpd - line printer daemon

SYNOPSIS

lpd

DESCRIPTION

Lpd is the line printer daemon which is run when the command _lpr_ (_1_) is typed. Only one daemon will be run at a time to prevent two or more items from being printed simultaneously. _Lpd_ prints a header, followed by a job.

FILES

/usr/spool/lpd/* Spool area for line printer

SEE ALSO

lpr(1)

NAME
lpr - line printer spooler

SYNOPSIS
lpr [name ...]

DESCRIPTION
Lpr causes the named files to be queued for printing. If no files are named, the standard input is read.

FILES
/usr/spool/lpd/*	spool area
/usr/lib/lpd	printer daemon

SEE ALSO
pr(1)

NAME

ls - list contents of directory

SYNOPSIS

ls [-lACFRabcdfgilmnqrstux] name ...

DESCRIPTION

For each directory argument, _ls_ lists the contents of the directory; for each file argument, _ls_ repeats the file name(s) and any other information requested with the _ls_ options. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The format chosen depends on whether the output is going to a teletype, and may also be controlled by option flags. The default format for a teletype is to list the contents of directories in multi-column format, with the entries sorted down the columns. (Files which are not the contents of a directory being interpreted are always sorted across the page rather than down the page in columns. This is because the individual file names may be arbitrarily long.) Files are listed first, and each directory being listed is labeled with its pathname, when two or more directory listings are requested. If the standard output is not a teletype, the default format is to list one entry per line. Finally, there is a stream output format in which files are listed across the page, separated by ',' characters. The -m flag enables this format.

There are numerous options:

-l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

-t Sort by time modified (latest first) instead of by name, as is normal.

-a List all entries; usually '.' and '..' (standing for the current directory and its immediate parent, respectively) are suppressed.

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

-d If argument is a directory, list only its name, not its contents (mostly used with -l to get status on directory).

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-u Use time of last access instead of last modification for sorting (-t) or printing (-l).

-c Use time of file creation for sorting (-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.

-m force stream output format.

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

-C force multi-column output, e.g. to a file or a pipe.

-q force printing of non-graphic characters in file names as the character '?'; this normally happens only if the output device is a teletype.

-b force printing of non-graphic characters to be in the \ddd notation, in octal.

-x force columnar printing to be sorted across rather than down the page; this is the default if the last character of the name the program is invoked with is an 'x' (for exaple, by linking /bin/ls to /bin/lx).

-F cause directories to be marked with a trailing '/' and executable files to be marked with a trailing '*'; this is the default if the last character of the name the program is invoked with is a 'f' (for example, by linking /bin/ls to /bin/lf).

-R recursively list subdirectories encountered.

The mode printed under the -l (long) option contains 11 characters which are interpreted as follows: (see also (chmod(1)).
The first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
m if the entry is a multiplexor-type character special file;
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as s if the file has set-group-ID mode; likewise the user-execute permission character is given as s if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '-') is t if the 1000 bit of the mode is on. See chmod(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user and group ID's given in "ls -l".

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

Column widths choices are poor for terminals which can tab.

NAME
mail - send or receive mail among users

SYNOPSIS
mail person ...
mail [-r] [-q] [-p] [-f file]

DESCRIPTION
Mail with no argument prints a user's mail, message-by-message, in last-in, first-out order; the optional argument -r causes first-in, first-out order. If the -p flag is given, the mail is printed with no questions asked; otherwise, for each message, mail reads a line from the standard input to direct disposition of the message.

newline
Go on to next message.

d Delete message and go on to the next.

p Print message again.

\- Go back to previous message.

s [file] ...
Save the message in the named files ('mbox' default).

w [file] ...
Save the message, without a header, in the named files ('mbox' default).

m [person] ...
Mail the message to the named persons (yourself is default).

EOT (control-D)
Put unexamined mail back in the mailbox and stop.

q Same as EOT.

x Exit, without changing the mailbox file.

!command
Escape to the Shell to do command.

? Print a command summary.

An interrupt stops the printing of the current letter. The optional argument -q causes mail to exit after interrupts without changing the mailbox.

When persons are named, mail takes the standard input up to an end-of-file (or a line with just '.') and adds it to each person's "mail" file. The message is preceded by the sender's name and a postmark. Lines that look like postmarks are prepended with '>'. A _person_ is usually a user name recognized by _login_(1). To denote a recipient on a remote system, prefix _person_ by the system name and exclamation mark.

The -f option causes the named file, e.g. 'mbox', to be printed as if it were the mail file.

Each user owns his own mailbox, which is by default generally readable but not writeable. The command does not delete an empty mailbox nor change its mode, so a user may make it unreadable if desired.

When a user logs in he is informed of the presence of mail.

EXAMPLE

```
mail karen
```

accepts whatever message is typed up to an EOF. Karen will be notified that she has mail when she next logs in.

If you want to read mail that has been sent to you, simply type

```
mail
```

FILES

/usr/spool/mail/*	mailboxes
/etc/passwd	to identify sender and locate persons
mbox	saved mail
/tmp/ma*	temp file
dead.letter	unmailable text

SEE ALSO

write(1)

BUGS

There is a locking mechanism intended to prevent two senders from accessing the same mailbox, but it is not perfect and races are possible.

NAME

make - maintain program groups

SYNOPSIS

make [-f makefile] [option] ... file ...

DESCRIPTION

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no -f option is present, "makefile" and "Makefile" are tried in order. If *makefile* is "-", the standard input is taken. More than one -f option may appear

Make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon, and all following lines that begin with a tab, are shell commands to be executed to update the target. If a name appears on the left of more than one "colon" line, then it depends on all of the names on the right of the colon on those lines, but only one command sequence may be specified for it. If a name appears on a line with a double colon :: then the command sequence following that line is performed only if the name is out of date with respect to the names to the right of the double colon, and is not affected by other double colon lines on which that name may appear.

Two special forms of a name are recognized. A name like *a*(*b*) means the file named *b* stored in the archive named *a*. A name like *a*((*b*)) means the file stored in archive *a* containing the entry point *b*.

Sharp and newline surround comments.

The following makefile says that "pgm" depends on two files "a.o" and "b.o", and that they in turn depend on ".c" files and a common file "incl".

```
pgm: a.o b.o
      cc a.o b.o -lm -o pgm
a.o: incl a.c
      cc -c a.c
b.o: incl b.c
      cc -c b.c
```

Makefile entries of the form

```
string1 = string2
```

are macro definitions. Subsequent appearances of $(*string1*) are replaced by *string2*. If *string1* is a single character, the parentheses are optional.

Make infers prerequisites for files for which *makefile* gives no construction commands. For example, a ".c" file may be inferred as prerequisite for a ".o" file and be compiled to produce the ".o" file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o -lm -o pgm
a.o b.o: incl
```

Prerequisites are inferred according to selected suffixes listed as the "prerequisites" for the special name ".SUFFIXES"; multiple lists accumulate; an empty list clears what came before. Order is significant; the first possible name for which both a file and a rule as described in the next paragraph exist is inferred. The default list is

```
.SUFFIXES: .out .o .c .e .r .f .y .l .s .p
```

The rule to create a file with suffix *s2* that depends on a similarly named file with suffix *s1* is specified as an entry for the "target" *s1s2*. In such an entry, the special macro `$*` stands for the target name with suffix deleted, `$@` for the full target name, `$<` for the complete list of prerequisites, and `$?` for the list of prerequisites that are out of date. For example, a rule for making optimized ".o" files from ".c" files is

```
.c.o: ; cc -c -O -o $@ $*.c
```

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, "CFLAGS" is used for *cc*(1) options, and "LFLAGS" and "YFLAGS" for *lex* and *yacc*(1) options.

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the special target ".SILENT" is in *makefile*, or the first character of the command is "@".

Commands returning nonzero status (see *intro*(1)) cause *make* to terminate unless the special target ".IGNORE" is in *makefile* or the command begins with <tab><hyphen>.

Interrupt and quit cause the target to be deleted unless the target depends on the special name ".PRECIOUS".

Other options:

-i Equivalent to the special entry ".IGNORE:".

-k When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.

-n Trace and print, but do not execute the commands needed to update the targets.

-t Touch, i.e. update the modified date of targets, without executing any commands.

-r Equivalent to an initial special entry ".SUFFIXES:" with no list.

-s Equivalent to the special entry ".SILENT:".

FILES

makefile	default input commnds to _make_
Makefile	default alternate input commands to _make_

SEE ALSO

sh(1), touch(1)
S. I. Feldman _Make - A Program for Maintaining Computer Programs_

BUGS

Some commands return nonzero status inappropriately. Use -i to overcome the difficulty.
Commands that are directly executed by the shell, notably _cd_(1), are ineffectual across newlines in _make_.

NAME

makekey - generate encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e. to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, upper- and lower-case letters, \`.' and \`/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

Makekey is intended for programs that perform encryption (e.g. *ed*(1) and *crypt*(1)). Usually its input and output will be pipes.

SEE ALSO

crypt(1), ed(1)

NAME

man - print sections of this manual

SYNOPSIS

man [option ...] [chapters] title ...

DESCRIPTION

Man locates and prints the section of this manual named *title* in the specified *chapters*. (In this context, the word 'page' is often used as a synonym for 'section'.) The *title* is entered in lower case. The *chapter* numbers do not need a letter suffix. If no *chapters* are specified, the whole manual is searched for *title* and the first occurrence of it is printed.

From the CRT, a call to *man* with a title or topic name prints out the specified manual section in *nroff'ed* form on the CRT, automatically piping it through *more*.

Manual sections may be preprocessed by *nroff* and put in *cat* files, as in /usr/man/cat?/*
If necessary, specific options may be added to print out manual sections in the desired form on the desired medium.

Options and their meanings are:

-t Phototypeset the section using *troff*(1).

-n Print the section on the standard output using *nroff*(1).

-k Display the output on a Tektronix 4014 terminal using *troff*(1) and *tc*(1).

-e Appended or prefixed to any of the above causes the manual section to be preprocessed by *neqn* or *eqn*(1); -e alone means -te.

-w Print the path names of the manual sections, but do not print the sections themselves.

-m Pipe the manual sections through *more*.

-u Pipe the manual sections through *ul*.

-s Remove extra blank lines as if the sections were being piped through *ssp*.

-d If one only has an nroff'able copy then use *deroff* instead of *nroff*.

-f stop after the first file is found.

-p Look for the files in the current directory.

- A single - will reset all options.

(default)
Copy an already formatted manual section to the terminal, or, if none is available, act as -n. It may be necessary to use a filter to adapt the output to the particular terminal's characteristics.

If the output device is a terminal then the f, s, m and u options will be set unless turned off by the - option.

Options and _chapters_ may be changed before each _title_.

EXAMPLE

For example:

```
man getc
```

would print out the manual page on "getc" from Section 3.

```
man 2 chmod
```

would print out the section 2 chapter on _chmod_, which comes from /usr/man/man2/chmod.2.
If the "2" had not been specified in the request, the section 1 chapter on _chmod_ would have been retrieved, since that would have been the first chapter on _chmod_ that _man_ found.

FILES

/usr/man/man?/*	for nroff manual sections
/usr/man/cat?/*	for preprocessed manual sections
/bin/cast	concatenate and print
/bin/ul	convert underline for terminals
/bin/ssp	remove extra blank line

SEE ALSO

nroff(1), eqn(1), tc(1), man(7)

BUGS

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

Some of the fancy options have not been fully tested or debugged.

NAME

mesg - permit or deny messages

SYNOPSIS

mesg [n] [y]

DESCRIPTION

Mesg with argument n forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *Mesg* with argument y reinstates permission. All by itself, *mesg* reports the current state without changing it.

EXAMPLE

```
mesg y
```

changes the permission to "yes", and the system reports:
Is Yes; Was No
or whatever the current and former state of your message permission is in fact.

FILES

/dev/tty*

SEE ALSO

write(1)

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME

mkdir - make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777. (see *chmod*(*1*)). Standard entries, '.', for the directory itself, and '..' for its parent, are made automatically. These and other directories beginning with . are not visible in listings unless you use the -a option to *ls*.

Mkdir requires write permission in the parent directory.

Mkdir runs as a "setuid" root program.

EXAMPLE

```
mkdir dirjohn
```

creates a directory of that name as a subdirectory of the directory you are in at the time you employ the command.

SEE ALSO

rm(1), rmdir(1)

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made. Otherwise it prints a diagnostic and returns nonzero.

NAME

mkfs - construct a file system

SYNOPSIS

mkfs special size [m n]
mkfs special proto

DESCRIPTION

Mkfs constructs a file system by writing on the special file _special_. In the first form of the command a numeric size is given and _mkfs_ builds a file system with a single empty directory on it. The number of i-nodes is calculated as a function of the filesystem size. _m_ is an interleave factor for building the freelist and _n_ is a modulo for _m_. See the example for usage.

N.B.: All filesystems should have a _lost+found_ directory for _fsck_(1M); this should be created for each file system by running _mklost+found_(1M) in the root directory of a newly created file system, after the file system is first mounted.

In bootstrapping, the second form of _mkfs_ is sometimes used. In this form, the file system is constructed 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 sector zero as the bootstrap program. The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of i-nodes in the i-list. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters -bcd specify regular, block special, character special and directory files respectively.) The second character of the type is either u or - to specify set-user-id mode or not. The third is g or - for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see _chmod_(1).

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

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

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

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

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
      $
$
```

EXAMPLE

```
mkfs /dev/fd0 2000 7 50
```

makes a file system in which 2000 is the total size of the file system to be put on _/dev/fd0_; 7 is a sector interleave number which is used to stagger the disk blocks for more rapid reading, every 7 blocks, and 50 is a modulo operator that forces the sector interlace number first to allocate all blocks in the first 50 sectors, then the next 50, etc.

NOTE: The proper selection of the _m_ and _n_ parameters can improve disk efficiency. Disks which have full or partial track buffering should specify a _m_ and _n_ of 1 and 1. _m_ and _n_ for other disks must be determined by trial and error as the disk latency is related to rotational latency and cpu speed.

SEE ALSO

filsys(5), dir(5), fsck(1M), mklost+found(1M)

BUGS

The default is 3500, which is probably not useful on any disk.
There should be some way to specify links.
There should be some way to specify bad blocks.
Should make _lost_+_found_ automatically.

NAME

mklost+found - make a lost+found directory for fsck

SYNOPSIS

mklost+found

DESCRIPTION

A directory _lost+found_ is created in the current directory and a number of empty files are created therein and then removed so that there will be empty slots for _fsck_(1M). This command should be run immediately after first mounting and changing directory to a newly created file system. For small file systems, it is sufficient (and much faster) to simply make a lost+found directory. Up to 30 files can be recovered in it.

SEE ALSO

fsck(1M), mkfs(1M)

BUGS

Should be done automatically by _mkfs_.

NAME

mknod - build special file

SYNOPSIS

mknod name [c] [b] major minor

DESCRIPTION

Mknod makes a special file. The first argument is the _name_ of the entry. The second is b if the special file is block-type (disks, tape) or c if it is character-type (other devices). The last two arguments are numbers specifying the _major_ device type and the _minor_ device (e.g. unit, drive, or line number).

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

EXAMPLE

```
mknod /dev/tty4 c 3 4
```

would create file /dev/tty4 as a character special device with major number 3 and minor number 4.

SEE ALSO

mknod(2)

NAME

mkstr - create an error message file by massaging C source

SYNOPSIS

mkstr [-] messagefile prefix file ...

DESCRIPTION

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

Mkstr will process each of the specified *files*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. A typical usage of *mkstr* would be

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file *mkstr* keys on the string `error("' in the input stream. Each time it occurs, the C string starting at the `"' is placed in the message file followed by a new-line character and a null character; the null character terminates the message so it can be easily used when retrieved, the new-line character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file which can be used to retrieve the message, i.e.:

```
char  efilname[] =  "/usr/lib/pi_strings";
int   efil = -1;

error(a1, a2, a3, a4)
{
      char buf[256];

      if (efil < 0) {
            efil = open(efilname, 0);
            if (efil < 0) {
oops:
                  perror(efilname);
                  exit(1);
            }
      }
      if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
            goto oops;
      printf(buf, a2, a3, a4);
```

```
}
```

The optional - causes the error messages to be placed at the end of the specified message file for recompiling part of a large _mkstr_ed program.

SEE ALSO

lseek(2), xstr(1)

AUTHORS

Bill Joy and Charles Haley

BUGS

All the arguments except the name of the file to be processed are unnecessary.

NAME

more - file perusal filter for crt viewing

SYNOPSIS

more [-dfl*n*] [+*linenumber* | +/*pattern*] [name ...]

DESCRIPTION

More is a filter which allows examination of a continuous text one screenful at a time on a CRT terminal. It normally pauses after each screenful, printing --More-- at the bottom of the screen.

If the user then types a carriage return, one more line is displayed. If the user hits a space, another screenful is displayed. If a space is preceded by an integer, that number of lines is printed. If the user hits d or control-D, 11 more lines are displayed (a 'scroll').

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

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

The following options are available:

-n is an integer which is the size (in lines) of the window which *more* will use instead of the default.

-d causes *more* to prompt the user with the message "Hit space to continue, Rubout to abort" at the end of each screenful.

-l causes *more* not to treat ^L (form feed) specially. If this option is not given, *more* will pause after any line that contains a ^L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

+*linenumber*
option causes *more* to start up at *linenumber*

+/*pattern*
causes *more* to start up two lines before the line containing the regular expression *pattern*.

Once inside *more*, other sequences may be typed when *more* pauses. The sequences and their effects are as follows (*i* is an optional integer argument, defaulting to 1) :

*i*z same as typing a space except that *i*, if present, becomes the new window size.

*i*s skip *i* lines and print a screenful of lines

*i*f skip *i* screenfuls and print a screenful of lines

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

*i*p skip to the *i*-th previous file given in the command line. If this command is given in the middle of printing out a file, then *more* goes back to the beginning of the file. If *i* doesn't make sense, *more* skips back to the first file. If *more* is not reading from a file, the bell is rung and nothing else happens.

q Exit from more.

i/expr
: search for the *i*-th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

' (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

!command
: invoke a shell with *command*.

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

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\). *More* will stop sending output, and will display the usual --More-- prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

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

If the standard output is not a teletype, then *more* acts just like *cat*, except that a header is printed before each file (if there is more than one).

EXAMPLE

A sample usage of *more* in previewing *nroff* output would be

```
nroff -ms +2 doc.n | more
```

AUTHOR

Eric Shienbrood

FILES

/etc/termcap	Terminal data base
/usr/lib/more.help	Help file

BUGS

The function of *more* should be done optionally by the teletype driver in the system, akin to the ''more'' feature of the ITS systems at MIT.

NAME

mount, umount - mount and dismount file system

SYNOPSIS

mount [special name [-r]]

umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device _special_. The file _name_ must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root. The optional argument -r indicates that the file system is to be mounted read-only.

Umount announces to the system that the removable file system previously mounted on device _special_ is to be removed.

These commands maintain a table of mounted devices in /_etc_/_mtab_. This table is only a reflection of what the mount and umount commands think is mounted, not what is actually mounted. If invoked without an argument, _mount_ prints the table.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

FILES

/etc/mtab mount table

SEE ALSO

mount(2)

BUGS

Mounting file systems full of garbage will crash the system.
Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

NAME

mv - move or rename files

SYNOPSIS

mv file1 file2

mv file ... directory

DESCRIPTION

Mv moves (changes the name of) *file1* to *file2*.

You can only *mv* files that you own or for which you have write permission. (see *chmod*(*1*)).

If *file2* already exists, it is removed before *file1* is moved. If *file2* exists and has a mode which forbids writing, *mv* prints the mode (see *chmod*(2)) and looks for a "y" from the standard input, which says "yes" to the move. This could be a "y" entered interactively or one at the beginning of the next line seen by the command interpreter. If no "y" is found, *mv* exits.

if *file2* does not exist, it is created for the move.

In the second form, one or more *files* are moved to the named *directory* with their original file-names.

Mv refuses to move a file onto itself.

EXAMPLE

```
mv /a/unisoft/bin/file1 /b/clara/file2
```

removes file1 from the first directory and stores it as file2 in the second directory.

FILES

/bin/cp   to do copy

SEE ALSO

cp(1), ln(1)

BUGS

If *file1* and *file2* lie on different file systems, *mv* must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

NAME

ncheck - generate names from i-numbers

SYNOPSIS

ncheck [-i numbers] [-a] [-s] [filesystem]

DESCRIPTION

N.B.: For most normal file system maintenance, the function of _ncheck_ is subsumed by _fsck_(1M).

Ncheck with no argument generates a pathname vs. i-number list of all files on a set of default file systems. Names of directory files are followed by '/.'. The -i option reduces the report to only those files whose i-numbers follow. The -a option allows printing of the names '.' and '..', which are ordinarily suppressed. The -s option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

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

EXAMPLE

```
ncheck /dev/rdisk1
```

will report the pathnames and i-numbers of files on the specified device.

SEE ALSO

sort(1), dcheck(1M), fsck(1M), icheck(1M)

DIAGNOSTICS

When the filesystem structure is improper, "??" denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

NAME

newgrp - log in to a new group

SYNOPSIS

newgrp group

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login*(1). The same person remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

A password is demanded if the group has a password and the user himself does not.

Newgrp is known to the shell, which executes it directly without a fork.

FILES

/etc/group, /etc/passwd

SEE ALSO

login(1), group(5)

NAME

nice - run a command at low priority

nohup - run a command immune to hangups (sh only)

SYNOPSIS

nice [-*number*] command [arguments]

nohup command [arguments]

DESCRIPTION

Nice executes *command* with low scheduling priority. In both *sh* and *csh*, priority numbers go from 0 (the highest priority) to 120 (the lowest priority). The normal priority number for a process without *nice* is 20. The default with *nice* is 24.

However, the method of setting or changing a priority is quite different between *sh* and *csh*.

In *csh*, you set or change priorities by adding (+n) or if you are the super-user, subtracting (-n) numbers to lower or raise the priority, respectively.

In *sh*, on the other hand, the number argument (-n) is always taken as a parameter to be added to the default priority, which lowers it. The *number* (-n) argument increases the priority number from 20 to 20 + n , and lowers the priority accordingly. The total may not exceed 120.

Only the super-user may run commands with priority higher than normal by subtracting from the default priority, *e.g.*, "--10" in the Bourne shell (-sh), or "-10" in *csh*.

Nohup executes *command* immune to terminate (EOT, Control-D) signal from the controlling terminal. With *nohup*, the priority is automatically incremented by 5. *Nohup* should be used with processes running in background (with '&') in order to prevent it from responding to interrupts or stealing the input from the next person who logs in on the same terminal. In *csh*, processes run in background are automatically immune to hangups.

EXAMPLE

```
nice -5 nroff -ms filea fileb filec&
```

formats the three named files in the background with priority 25 (in *sh*),
OR
in *csh*, at priority 15.

FILES

nohup.out standard output and standard error file under nohup in sh(1).

SEE ALSO

csh(1), nice(2)

DIAGNOSTICS

Nice returns the exit status of the subject command.

To find out what the "nice" status of particular processes is, do a ps axl, and look in the "NICE" column. Stay aware of the shell you're in.

NAME

nm - print name list

SYNOPSIS

nm [-gnopru] [file ...]

DESCRIPTION

_Nm_ prints the name list (symbol table) of each object _file_ in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no _file_ is given, the symbols in "a.out" are listed.

Each symbol name is preceded by its value (blanks if undefined) and one of the letters U (undefined), A (absolute), T (text segment symbol), D (data segment symbol), B (bss segment symbol), C (common symbol), f file name. If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

Options are:

-g Print only global (external) symbols.

-n Sort numerically rather than alphabetically.

-o Prepend file or archive element name to each output line rather than only once.

-p Don't sort; print in symbol-table order.

-r Sort in reverse order.

-u Print only undefined symbols.

EXAMPLE

```
nm
```

prints the symbol list of _a.out_, the default output file for the C compiler.

FILES

/bin/sort to sort or merge files

SEE ALSO

ar(1), ar(5), a.out(5), stab(5)

NAME

nohup - run a command immune to hangups

SYNOPSIS

nohup command [arguments]

DESCRIPTION

nohup executes _command_ with hangups, quits, and interrupts all ignored. If the user does not specifically direct the output from a command, the output is directed to the file nohup.out in the current directory. If the current directory is not writeable, the output is redirected to $HOME/nohup.out.

EXAMPLE

The following _nohup_ call:

```
nohup  nroff -ms  docsfile | lpr
```

runs the _nroff_ command shown, immune to hangups, quits, and interrupts.

SEE ALSO

nice(1), _signal_(1)

NAME

troff, nroff - text formatting and typesetting

SYNOPSIS

nroff [option] ... [file] ...

DESCRIPTION

Nroff formats text in the named _files_ for typewriter-like devices. See also _troff_(_1_). The full capabilities of _nroff_ and _troff_ are described in the _Nroff_/_Troff_ _User's_ _Manual_.

If no _file_ argument is present, the standard input is read. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input.

The options, which may appear in any order so long as they appear _before_ the files, are:

-o_list_ Print only pages whose page numbers appear in the comma-separated _list_ of numbers and ranges. A range _N_-_M_ means pages _N_ through _M_; an initial -_N_ means from the beginning to page _N_; and a final _N_- means from _N_ to the end.

-n_N_ Number first generated page _N_.

-s_N_ Stop every _N_ pages. _Nroff_ will halt prior to every _N_ pages (default _N_=1) to allow paper loading or changing, and will resume upon receipt of a newline.

-m_name_ Prepend the macro file /usr/lib/tmac/tmac._name_ to the input _files_.

-r_aN_ Set register _a_ (one-character) to _N_.

-i Read standard input after the input files are exhausted.

-q Invoke the simultaneous input-output mode of the rd request.

-T_name_ Prepare output for specified terminal. Known _names_ are 37 for the (default) Teletype Corporation Model 37 terminal, tn300 for the GE TermiNet 300 (or any terminal without half-line capability), 300S for the DASI-300S, 300 for the DASI-300, and 450 for the DASI-450 (Diablo Hyterm).

-e Produce equally-spaced words in adjusted lines, using full terminal resolution.

-h Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

EXAMPLE

```
nroff -s4 -me filea
```

will _nroff_ the named file using the -me macro package, stopping every 4 pages.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables for _nroff_

SEE ALSO

J. F. Ossanna, _Nroff/Troff user's manual_
B. W. Kernighan, _A TROFF Tutorial_
troff(1), eqn(1), tbl(1), ms(7), me(7), man(7), col(1)

NAME

num - number lines

SYNOPSIS

num [file ...]

DESCRIPTION

The lines in the specified files, or the standard input, are copied to the standard output preceded by line numbers. Tabs remain aligned in the output as the lines are printed preceded by the number blank padded to six digits and then 2 spaces.

EXAMPLE

```
num filea > fileb
```

will number the lines of _filea_ and send the output to _fileb_.

SEE ALSO

cat(1), pr(1)

NAME

od - octal dump

SYNOPSIS

od [-abcdoxDOXw] [file] [[+]offset[.][b]]

DESCRIPTION

_Od_ dumps _file_ in one or more formats as selected by the first argument. If the first argument is missing, -o is default. The meanings of the format argument characters are:

b Interpret bytes in octal.

c Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=\0, backspace=\b, formfeed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers.

d Interpret shorts (16 bit words) in decimal.

o Interpret shorts (16 bit words) in octal.

w Produce wide (132 column) output.

x Interpret shorts (16 bit words) in hex.

D Interpret longs (32 bit words) in decimal.

O Interpret longs (32 bit words) in octal.

X Interpret longs (32 bit words) in hex.

The _file_ argument specifies which file is to be dumped. If no file argument is specified, the standard input is used.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If '.' is appended, the offset is interpreted in decimal. If 'b' is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded '+'.

Dumping contiues until and end-of-file is received.

EXAMPLE

```
od -D filea +2
```

produces an octal dump of _filea_ divided up into 32-bit words expressed in decimal equivalents; with the dump starting point offset by 2 octal bytes.

SEE ALSO

adb(1)

NAME

passwd - change login password

SYNOPSIS

passwd [name]

DESCRIPTION

This command changes (or installs) a password associated with the user *name* (your own name by default).

The program prompts for the old password and then for the new one. The caller must supply both. The new password must be typed twice, to forestall mistakes.

It is suggested that new passwords be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password.

EXAMPLE

```
passwd
```

responds "Changing password for <username>, then asks for your password (once) and for the new password (twice).

FILES

/etc/passwd	
/etc/utmp	to ensure that user is logged in
/etc/ttys	to ensure that user is logged in

SEE ALSO

login(1), passwd(5)

NAME

pc - Pascal compiler

SYNOPSIS

pc [-o ofile] [-i] [-c] [-u] [-v] file ...

DESCRIPTION

pc, the PASCAL compiler, accepts a list of Pascal source files and various intermediate texts contained in the list of files specified by _file_ and puts the resulting executable object module in _a.out_ (but see the -o option, described below).

In order to understand the use of pc, the reader must first understand the steps which the compiler goes through in order to turn a Pascal source program into an executable object file.

The Pascal compiler generates several intermediate files on the way to generating the final executable file. The first phase of the compiler generates an intermediate file, of the same name as the source file, but with a .i suffix. This intermediate file is destined for processing by the code generator.

The code generator is the second phase of the process. The output of the code generator is a file with the same name as the source file, but with a suffix of .obj. The *.obj file is the input to the next phase, called ulinker.

The ulinker phase of the compilation process converts the .obj file into a UNIX-style object file with a .o suffix. This file can then be processed by the UNIX loader utility, ld.

Finally, the ld utility produces the final executable code file.

When using pc, any combination of Pascal source files (each having a .for suffix) can be combined with Pascal or FORTRAN intermediate files (each having a .i suffix), Pascal or FORTRAN object code files (each having a .obj suffix), and UNIX object files (each having a .o suffix). When the compilation completes successfully, the result of the combination of all those files is placed in the file _a.out_ or in the file specified by the -o option.

The -o option, if given, specifies that the file _ofile_ (runnable file) whose name follows the option is the file to receive the final executable code. If the -o option is not specified, the resultant executable file is placed in the file _a.out_.

If the -i option is given, the Pascal intermediate code (the result of running _/lib/pascal_) is placed in a file of the same name as the source file, but with a suffix of .i appended. The compilation then terminates.

If the -c option is given, the Pascal unlinked object code (the result of running */lib/code*) is placed in a file of the same name as the source file, but with a suffix of .obj appended. The compilation then terminates.

If the -u option is given, the linked object code (the result of running /*lib*/*ulinker*) is placed in a file of the same name as the source file, but with a suffix of .o appended. The compilation then terminates.

The -v (for verbose) option makes pc display a running progress report as it compiles.

If only one *file* argument is supplied on the command line, then all the intermediate files (.i, .obj, .o) are removed at the end of the compilation. If multiple *file* arguments are typed on the command line, any existing intermediate files are not removed.

EXAMPLES

```
pc progl.pas
```

compiles *progl.pas* and puts the resulting object module in *a.out*.

```
pc -o frammis prog2.pas prog3.obj
```

compiles the Pascal program called *prog2.pas* and links the result with the object file *prog3.obj*. The result of the compilation is placed in the output file called *frammis*.

FILES

*.pas	Pascal source
*.i	Intermediate code
*.obj	Compiled unlinked pc object
*.o	Compiled unlinked UNIX object
/lib/paslib.obj	
/lib/pascal	
/lib/code	
/lib/ulinker	
/lib/pascterrs	

SEE ALSO

"User Documentation Update for UniSoft Pascal and FORTRAN".

NAME

pr - print file

SYNOPSIS

pr [option] ... [file] ...

DESCRIPTION

Pr produces a printed listing of one or more *files*. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, *pr* prints its standard input.

Options apply to all following files but may be reset between files:

-*n* Produce *n*-column output.

+*n* Begin printing with page *n*.

-h Take the next argument as a page header.

-w*n* For purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72.

-f Use formfeeds instead of newlines to separate pages. A formfeed is assumed to use up two blank lines at the top of a page. (Thus this option does not affect the effective page length.)

-l*n* Take the length of the page to be *n* lines instead of the default 66.

-t Do not print the 5-line header or the 5-line trailer normally supplied for each page.

-s*c* Separate columns by the single character *c* instead of by the appropriate amount of white space. A missing *c* is taken to be a tab.

-m Print all *files* simultaneously, each in one column,

Inter-terminal messages via *write*(1) are forbidden during a *pr*.

EXAMPLE

```
pr -t -m filea fileb filec
```

will print out the three files simultaneously, each in one column, without headers.

FILES

/dev/tty? to suspend messages.

SEE ALSO

cat(1)

DIAGNOSTICS

There are no diagnostics when _pr_ is printing on a terminal.

NAME

printenv - print out the environment

SYNOPSIS

printenv

DESCRIPTION

Printenv prints out the values of the variables in the environment.

The environment variable names are:

HOME path name of home directory.

PATH search path for binary programs

TERM type of terminal used

SHELL the shell present at login.

EXAMPLE

```
printenv
```

prints the defined variables in the environment.

SEE ALSO

csh(1), sh(1), environ(5)

NAME
prof - display profile data

SYNOPSIS
prof [-a] [-l] [-z] [-_low_ [-_high_]]] [a.out [mon.out ...]]

DESCRIPTION
Prof interprets the file 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 profile file (_mon.out_ default). 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 more than one profile file is specified, the output represents the sum of the profiles.

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 profile file to be produced automatically.

Options are:

-a all symbols are reported rather than just external symbols.

-l the output is sorted by symbol value.

-z routines which have zero usage (as indicated by call counts and accumulated time) are nevertheless printed in the output.

FILES
mon.out for profile
a.out for namelist
mon.sum for summary profile

SEE ALSO
cc(1), monitor(3), profil(2)

BUGS
Beware of quantization errors.

NAME

prs - print an SCCS file

SYNOPSIS

prs [-d[dataspec]] [-r[SID]] [-e] [-l] [-a] files

DESCRIPTION

Prs prints, on the standard output, parts or all of an SCCS file (see *sccsfile*(5)) in a user supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with s.), and unreadable files are silently ignored. If a name of - is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of *keyletter* arguments, and file names.

All the described *keyletter* arguments apply independently to each named file:

-d[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *DATA KEYWORDS*) interspersed with optional user supplied text.

-r[*SID*] Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

-e Requests information for all deltas created *earlier* than and including the delta designated via the -r keyletter.

-l Requests information for all deltas created *later* than and including the delta designated via the -r keyletter.

-a Requests printing of information for both removed, that is, delta type = *R*, (see *rmdel*(1)) and existing, that is, delta type = *D*, deltas. If the -a keyletter is not specified, information for existing deltas only is provided.

DATA KEYWORDS

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(5)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by *prs* consists of: (1) the user supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User supplied text is any text other than recognized data keywords. A tab is specified by \t and carriage return/new-line is specified by \n.

TABLE 1. SCCS Files Data Keywords

Keyword	Data Item	File Section	Value	Format
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	D or R	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th:::Tm:::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	yes or no	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	yes or no	S
:BF:	Branch flag	"	yes or no	S
:J:	Joint edit flag	"	yes or no	S
:LK:	Locked releases	"	:R:...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	yes or no	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(1) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(1) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(1) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S

:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

```
prs -d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
prs -d"Newest delta for pgm :M:: :I: Created :D: By :P:" -r s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case*:

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
b178-12345
b179-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only keyletter argument allowed to be used with the *special case* is the -a keyletter.

FILES

/tmp/pr?????
/etc/mtab mounted file system table

SEE ALSO

admin(1), delta(1), get(1), help(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

ps - process status

SYNOPSIS

ps [acgklrtuwx# [namelist]]

DESCRIPTION

Ps prints information about active processes. To get a complete printout on the console or lpr, use "ps axlw" For a quick snapshot of system activity, "ps au" is recommended. A hyphen may precede options with no effect. The following options may be specified.

a
: asks for information about all processes with terminals (ordinarily only one's own processes are displayed).

c
: causes only the *comm* field to be displayed instead of the arguments. (The comm field is the tail of the path name of the file the process last exec'ed.) This option speeds up *ps* somewhat and reduces the amount of output. It is also more reliable since the process can't scribble on top of it.

g
: Asks for all processes. Without this option, *ps* only prints "interesting" processes. Processes are deemed to be uninteresting if they are process group leaders, or if their arguments begin with a '-'. This normally eliminates shells and getty processes.

k
: causes the file /*usr*/*sys*/*core* is used in place of /*dev*/*kmem* and /*dev*/*mem*. This is used for postmortem system debugging.

l
: asks for a long listing. The short listing contains the user name, process ID, tty, the cumulative execution time of the process and an approximation to the command line.

r
: asks for "raw output". A non-human readable sequence of structures is output on the standard output. There is one structure for each process, the format is defined by <psout.h>

t*tyname*
: restricts output to processes whose controlling tty is the specified ttyname (which should be specified as printed by *ps*, including *t*? for processes with no tty). This option must be the last one given.

u
: A user oriented output is produced. This includes the name of the owner of the process, process id, nice value, size, tty, cpu time used, and the command.

w
: tells *ps* you are on a wide terminal (132 columns). *Ps* normally assumes you are on an 80 column terminal. This information is used to decide how much of long commands to print. The w option may be repeated, e.g. ww, and the entire command, up to 128 characters, will be printed without regard to terminal width.

x asks even about processes with no terminal.

\# A process number may be given, (indicated here by #), in which case the output is restricted to that process. This option must also be last.

A second argument tells ps where to look for core if the k option is given, instead of /usr/sys/core. A third argument is the name of a swap file to use instead of the default /dev/swap. If a fourth argument is given, it is taken to be the file containing the system's namelist. Otherwise, "/unix" is used.

The output is sorted by tty, then by process ID.

The long listing is columnar and contains

F Flags associated with the process. The flags are defined in /usr/include/sys/proc.h, and include:

SLOAD	000001	in core
SSYS	000002	swapper process
SLOCK	000004	process being swapped out
SSWAP	000008	save area flag
STRC	000010	process is being traced

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

UID The user id of the process owner.

PID The process ID of the process.

PPID The process ID of the parent process.

CPU Processor utilization for scheduling.

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

NICE Used in priority computation.

ADDR The memory address of the process if resident, otherwise the disk address.

SZ The size in blocks of the memory image of the process.

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

TTY The controlling tty for the process.

TIME The cumulative execution time for the process.

COMMAND
The command and its arguments.

A process that has exited and has a parent, but has not yet been waited for by the parent is marked <defunct>. _Ps_ makes an educated guess as to the file name and arguments given when the process was created by examining 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/kmem	kernel memory
/dev/swap	swap device
/core	core file
/dev	searched to find swap device and tty names
/dev/mem	physical memory
/usr/sys/core	for postmortem system debugging

SEE ALSO

kill(1)

BUGS

Things can change while _ps_ is running; the picture it gives is only a close approximation to reality.

Some processes, typically those in the background, are printed with null or garbaged arguments, even though the process has not swapped. (Sometimes ps even loses on its own arguments!) In these cases, the name of the command is printed in parentheses.

NAME

pstat - print system facts

SYNOPSIS

pstat [-aixptuf] [suboptions] [file]

DESCRIPTION

Pstat interprets the contents of certain system tables. If *file* is given, the tables are sought there, otherwise in */dev/mem*. The required namelist is taken from */unix*. Options are

-a Under -p, describe all process slots rather than just active ones.

-i Print the inode table with the these headings:

LOC The core location of this table entry.

FLAGS Miscellaneous state variables encoded thus:

- L locked
- U update time *filsys*(5)) must be corrected
- A access time must be corrected
- M file system is mounted here
- W wanted by another process (L flag is on)
- T contains a text file
- C changed time must be corrected

CNT Number of open file table entries for this inode.

DEV Major and minor device number of file system in which this inode resides.

INO I-number within the device.

MODE Mode bits, see *chmod*(2).

NLK Number of links to this inode.

UID User ID of owner.

SIZ/DEV
Number of bytes in an ordinary file, or major and minor device of special file.

-x Print the text table with these headings:

LOC The core location of this table entry.

FLAGS Miscellaneous state variables encoded thus:

- T *ptrace*(2) in effect
- W text not yet written on swap device
- L loading in progress
- K locked
- w wanted (L flag is on)

DADDR Disk address in swap, measured in multiples of 512 bytes.

CADDR Core address, measured in multiples of core clicks (machine dependent).

SIZE Size of text segment, measured in multiples of core clicks (machine dependent).

IPTR Core location of corresponding inode.

CNT Number of processes using this text segment.

CCNT Number of processes in core using this text segment.

-p Print process table for active processes with these headings:

LOC The core location of this table entry.
S Run state encoded thus:
- 0 no process
- 1 waiting for some event
- 3 runnable
- 4 being created
- 5 being terminated
- 6 stopped under trace

F Miscellaneous state variables, or-ed together:
- 01 loaded
- 02 the scheduler process
- 04 locked
- 010 swapped out
- 020 traced
- 040 used in tracing
- 0100 locked in by _lock_(2).

PRI Scheduling priority, see _nice_(2).
SIGNAL
Signals received (signals 1-16 coded in bits 0-15),
UID Real user ID.
TIM Time resident in seconds; times over 127 coded as 127.
CPU Weighted integral of CPU time, for scheduler.
NI Nice level, see _nice_(2).
PGRP Process number of root of process group (the opener of the controlling terminal).
PID The process ID number.
PPID The process ID of parent process.
ADDR If in core, the physical address of the "u-area" of the process measured in multiples of 64 bytes. If swapped out, the position in the swap area measured in multiples of 512 bytes.
SIZE Size of process image in multiples of 64 bytes.
WCHAN Wait channel number of a waiting process.
LINK Link pointer in list of runnable processes.
TEXTP If text is pure, pointer to location of text table entry.
CLKT Countdown for _alarm_(2) measured in seconds.

-t Print table for terminals (only DH11 and DL11 handled) with these headings:

RAW Number of characters in raw input queue.
CAN Number of characters in canonicalized input queue.
OUT Number of characters in putput queue.
MODE See _tty_(4).
ADDR Physical device address.

DEL Number of delimiters (newlines) in canonicalized input queue.
COL Calculated column position of terminal.
STATE Miscellaneous state variables encoded thus:
- W waiting for open to complete
- O open
- S has special (output) start routine
- C carrier is on
- B busy doing output
- A process is awaiting output
- X open for exclusive use
- H hangup on close

PGRP Process group for which this is controlling terminal.

-u print information about a user process; the next argument is its address as given by ps(1). The process must be in main memory, or the file used can be a core image and the address 0.

-f Print the open file table with these headings:

LOC The core location of this table entry.
FLG Miscellaneous state variables encoded thus:
- R open for reading
- W open for writing
- P pipe

CNT Number of processes that know this open file.
INO The location of the inode table entry for this file.
OFFS The file offset, see lseek(2).

FILES

/unix namelist
/dev/mem default source of tables

SEE ALSO

ps(1), stat(2), filsys(5)
K. Thompson, UNIX Implementation

NAME

ptx - permuted index

SYNOPSIS

ptx [option] ... [input [output]]

DESCRIPTION

Ptx generates a permuted index to file *input* on file *output* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of the page. *Ptx* produces output in the form:

    .xx "tail" "before keyword" "keyword and after" "head"

where .xx may be an *nroff* or *troff*(1) macro for user-defined formatting. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed at the middle of the page. *Tail* and *head*, at least one of which is an empty string "", are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line. When original text must be discarded, "/" marks the spot.

The following options can be applied:

-f Fold upper and lower case letters for sorting.

-t Prepare the output for the phototypesetter; the default line length is 100 characters.

-w *n* Use the next argument, *n*, as the width of the output line. The default line length is 72 characters.

-g *n* Use the next argument, *n*, as the number of characters to allow for each gap among the four parts of the line as finally printed. The default gap is 3 characters.

-o only
Use as keywords only the words given in the *only* file.

-i ignore
Do not use as keywords any words given in the *ignore* file. If the -*i* and -*o* options are missing, use /usr/lib/eign as the *ignore* file.

-b break
Use the characters in the *break* file to separate words. In any case, tab, newline, and space characters are always used as break characters.

-r Take any leading nonblank characters of each input line to be a reference identifier (as to a page or chapter) separate from the text of the line. Attach that identifier as a 5th field on each output line.

The index for this manual was generated using *ptx*.

FILES

/bin/sort
/usr/lib/eign

BUGS

Line length counts do not account for overstriking or proportional spacing.

NAME

put - puts a file onto a remote machine.

SYNOPSIS

put [-p port] [-s[SYSID]] fromfile [tofile]

DESCRIPTION

Put puts a file from a local machine onto a remote machine. The default port is /dev/tty0; the -_p_ _port_ option can be used to specify an alternate output port. The default system id is read from /etc/sys_id, specifying generic locations for the remote machine to look for the source; the -_s_[_SYSID_] option specifies an alternate system id.

fromfile The local file name.

tofile The remote file name; if _tofile_ is null, tofile is defaulted to fromfile.

NOTES

This program requires the existence of the program _putll_ on the remote machine.

The -_s_ option requires the existence of the file /lib/MAKE.sys on the remote machine; the option is only useful to UniSoft Systems.

SEE ALSO

take(1)

AUTHOR

UniSoft Corporation of Berkeley.

NAME

pwd - working directory name

SYNOPSIS

pwd

DESCRIPTION

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

EXAMPLE

pwd

produces a pathname, such as /unisoft/sandy, indicating what directory you are currently in. By displaying the pathname of the directory you are currently in, *pwd* may show you that you are not where you thought you were. Being in an unexpected directory could bring on a sudden rash of error messages.

SEE ALSO

cd(1), csh(1)

NAME

reset - reset the teletype bits to a sensible state

SYNOPSIS

reset

DESCRIPTION

Reset sets the terminal to cooked mode, turns off cbreak and raw modes, turns on nl, and restores special characters that are undefined to their default values.

This is most useful after a program dies leaving a terminal in a funny state; you have to type ``<LF>reset<LF>'' to get it to work then to the shell, as <CR> often doesn't work; often none of this will echo.

It isn't a bad idea to follow *reset* with *tset*(1)

SEE ALSO

stty(1), tset(1)

BUGS

Doesn't set tabs properly; it can't intuit personal choices for interrupt and line kill characters, so it leaves these the old UNIX standards ^? (delete) for interrupt and @ for line kill.

It could well be argued that the shell should be responsible for insuring that the terminal remains in a sane state; this would eliminate the need for this program.

NAME

restor - incremental file system restore

SYNOPSIS

restor key [argument ...]

DESCRIPTION

Restor is used to read files from tape or disk that were dumped with the _dump_ command. The _key_ specifies what is to be done. _Key_ is one of the characters rxt and f.

f The first _argument_ after the "key" set of letters is the name of the dump device, whether tape or disk.

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

x Each file on the tape or disk named by an _argument_ is extracted. The file extracted is placed in a file with a numeric name supplied by _restor_ (actually the inode number). In order to keep the amount of tape or disk read to a minimum, the following procedure is recommended:

Mount volume 1 of the set of dump tapes or disks.

Type the _restor_ command.

Restor will announce whether or not it found the files, if given the number it will name the file, and rewind the tape or disk.

It then asks you to 'mount the desired tape or disk volume'. Type the number of the volume you choose. On a multivolume dump the recommended procedure is to mount the last through the first volume in that order. _Restor_ checks to see if any of the files requested are on the mounted tape or disk (or a later tape or disk, thus the reverse order).

If you are working with a single volume dump or the number of files being restored is large, respond to the query with '1' and _restor_ will read the tape or disks in sequential order.

If you have a hierarchy to restore you can use _dumpdir_(1M) to produce the list of names and a shell script to move the resulting files to their homes.

t Print the date the tape or disk was written and the date the filesystem was dumped from.

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

EXAMPLE

```
/etc/mkfs /dev/rrp0g 145673
restor rf /dev/rfdc1 /dev/rrp0g
```

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

default tape or disk unit varies with installation
rst*

SEE ALSO

dump(1M), mkfs(1M), dumpdir(1M)

DIAGNOSTICS

There are various diagnostics involved with reading the tape or disk 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 or disk, it may ask you to change tape or disks. Reply with a new-line when the next tape or disk has been mounted.

BUGS

There is redundant information on the tape or disk that could be used in case of tape or disk reading problems. Unfortunately, _restor_ doesn't use it.

NAME

rev - reverse lines of a file

SYNOPSIS

rev [file] ...

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

EXAMPLE

```
rev filea
```

reverses the characters in each line of *filea* and sends them to standard output.

NAME

rm - remove (unlink) files

SYNOPSIS

rm [-f] [-i] [-r] [-] file ...

DESCRIPTION

_Rm_ removes the entries for one or more files from a directory. If an entry was the last (or only) 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 is required. Paradoxically, you can remove a file with _rm_ even though you do not have permission to read or edit it.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with 'y' the file is deleted, otherwise the file remains.

No questions are asked and no errors are reported when the -f (force) option is given.

The -i option stands for interactive mode. The user is prompted by the name of the file. A response starting with y causes the file to be removed. Any other response is considered a no.

If a designated file is a directory, an error comment is printed unless the optional argument -r has been used. In that case, _rm_ recursively deletes the entire contents of the specified directory, and the directory itself, quickly and efficiently.

The null option - indicates that all the arguments following it are to be treated as file names. This allows the specification of file names starting with a minus.

EXAMPLE

```
rm -r dirname
```

will remove the entire contents of the named directory and all subdirectories, and finally the directory itself, with no questions asked.

FILES

/bin/rmdir to remove directory

SEE ALSO

rmdir(1), unlink(2)

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file ".." merely to avoid the antisocial consequences of inadvertently doing something like "rm -r .*".

NAME

rmcobol - COBOL compiler
runcobol - COBOL runtime interpreter

SYNOPSIS

rmcobol file [-d] [-c nn] [-l] [-n] [-o objfile] [-p nn] [-x]
runcobol file [-a] [-d] [-s nn..n]

DESCRIPTION

rmcobol is a single-pass compiler that generates intermediate code to be interpreted by the COBOL runtime interpreter "runcobol". When no options are specified, the compiler will put its output on the file named "cbl.out" in the current directory. The following options are accepted by the compiler:

-d Compile COBOL "Debug" source lines identified by "D" in column 7.

-c nn Set the maximum output line length for the listing file to nn. (The default is 80 characters.)

-e Generate 'Error Only' listing instead of full listing.

-l Output the listing to standard output.

-n Compile without generating an object file.

-o objfile Define an alternate output file "objfile".

-p nn Set the page size to nn number of lines.

-x Generate cross-reference listing; option valid only if the -e or -l option is specified.

runcobol is the COBOL runtime interpreter; it executes a compiled COBOL object program generated by rmcobol(1). The following runtime options are accepted by the interpreter:

-a Set automatic line-feed flag on.

-d Invoke the RMCOBOL Interactive Debug package.

-s nn..n Sets or resets value of SWITCHES in the COBOL program; where each "n" is a switch value, 0 for off, 1 for on, numbered 1 to 8, left to right.

For more detailed information, see RM/COBOL User's Guide.

EXAMPLES

rmcobol payroll -l -x

compiles the source program "payroll" in the current working directory, producing an object file "cbl.out"; a listing with cross references is written to the standard output file.

```
runcobol cbl.out -s 1011
```

loads and executes the COBOL object program cbl.out and sets the value of SWITCHES 1, 3, and 4 to "on", all others to "off".

FILES

/lib/rmcb1013	Cobol compile time modules
/lib/rmcb1113	
/lib/rmcb1213	
/lib/rmcb1313	
/lib/rmcb1413	

NAME

rmdel - remove a delta from an SCCS file

SYNOPSIS

rmdel -rSID files

DESCRIPTION

Rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must *not* be that of a version being edited for the purpose of making a delta (that is, if a *p-file* (see *get*(1)) exists for the named SCCS file, the SID specified must *not* appear in any entry of the *p-file*).

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

The exact permissions necessary to remove a delta are documented in the *Source Code Control System User's Guide*. Simply stated, they are either (1) if you make a delta you can remove it; or (2) if you own the file and directory you can remove a delta.

FILES

x-file (see *delta*(1))
z-file (see *delta*(1))

SEE ALSO

delta(1), get(1), help(1), prs(1), sccsfile(5).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

rmdir - remove an empty directory

SYNOPSIS

rmdir directory ...

DESCRIPTION

Rmdir removes an empty directory.

Once the directory has been removed, it is destroyed. Removal of a directory requires write permission in the parent directory.

Rmdir removes the named directories, which must be empty. *Rmdir* will otherwise report that the named directory is not empty.

Rmdir runs as a "setuid" root program.

EXAMPLE

```
rmdir dirname
```

removes the empty directory.

SEE ALSO

rm(1), unlink(2)

NAME
sact - print current SCCS file editing activity

SYNOPSIS
sact files

DESCRIPTION
Sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(1) with the -e option has been previously executed without a subsequent execution of *delta*(1). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of - is given, the standard input is read with each line being taken as the name of an SCCS file to be processed. The output for each named file consists of five fields separated by spaces.

Field 1 specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

Field 2 specifies the SID for the new delta to be created.

Field 3 contains the logname of the user who will make the delta (i.e. executed a *get* for editing).

Field 4 contains the date that get -e was executed.

Field 5 contains the time that get -e was executed.

SEE ALSO
delta(1), get(1), unget(1).

DIAGNOSTICS
Use *help*(1) for explanations.

NAME

sccsdiff - compare two versions of an SCCS file

SYNOPSIS

sccsdiff -rSID1 -rSID2 [-p] [-sn] files

DESCRIPTION

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

-r_SID_? _SID1_ and _SID2_ specify the deltas of an SCCS file that are to be compared. Versions are passed to _bdiff_(1) in the order given.

-p pipe output for each file through _pr_(1).

-s_n_ _n_ is the file segment size that _bdiff_ will pass to _diff_(1). This is useful when _diff_ fails due to a high system load.

FILES

/tmp/get????? Temporary files

SEE ALSO

bdiff(1), get(1), help(1), pr(1).
_Source Code Control System User's Guide_by L. E. Bonanni and C. A. Salemi.

DIAGNOSTICS

file :No_differences_ If the two versions are the same.
Use _help_(1) for explanations.

NAME

sed - stream editor

SYNOPSIS

sed [-n] [-e script] [-f sedfile] [file] ...

DESCRIPTION

Sed copies the named _files_ (standard input default) to the standard output, edited according to a script of commands. The -f option causes the script to be taken from file _sedfile_; these options accumulate. If there is just one -e option and no -f's, the flag -e may be omitted. The -n option suppresses the default output.

A _sedfile_ script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation _sed_ cyclically copies a line of input into a _pattern space_ (unless there is something left after a "D" command), applies in sequence all commands whose _addresses_ select that pattern space, and at the end of the script copies the pattern space to the standard output (except under -n) and deletes the pattern space.

An _address_ is either a _line number_, a decimal number that counts input lines cumulatively across files, or a "$" that addresses the last line of input, as in _ed_.

Address may also be a context address, using a "/regular expression/", in the style of _ed_(1)

In the notes below "pattern space" refers to those lines that match the line numbers or qualify because they contain the pattern specified in the context address.

Addresses may be modified in the following ways:

The escape sequence '\n' matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (That is, an address of "1,10" would mean that the commands should be performed on lines 1 through 10 inclusive). If the second address is a number less than or equal to the line number first selected, only one line is selected. Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function '!' (below).

An argument denoted _text_ consists of one or more lines, all but the last of which end with '\' to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an 's' command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument meaning the file to edit, or _rfile_, and/or the file to be written to, _wfile_ must terminate the command line and must be preceded by exactly one blank. Each _wfile_ that does not already exist is created before processing begins. There can be at most 10 distinct _wfile_ arguments.

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

(1)a\
text
: Append. Place _text_ on the output before reading the next input line.

(2)b _label_
: Branch to the ':' command bearing the _label_. If _label_ is empty, branch to the end of the script.

(2)c\
text
: Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place _text_ on the output. Start the next cycle.

(2)d Delete the pattern space. Start the next cycle.

(2)D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2)g Replace the contents of the pattern space by the contents of the hold space.

(2)G Append the contents of the hold space to the pattern space.

(2)h Replace the contents of the hold space by the contents of the pattern space.

(2)H Append the contents of the pattern space to the hold space.

(1)i\
text
: Insert. Place _text_ on the standard output.

(2)n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2)N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2)p Print. Copy the pattern space to the standard output.

(2)P Copy the initial segment of the pattern space through the first newline to the standard output.

(1)q Quit. Branch to the end of the script. Do not start a new cycle.

(2)r _rfile_
Read the contents of _rfile_. Place them on the output before reading the next input line.

(2)s/_regular expression_/_replacement_/_flags_
Substitute the _replacement_ string for instances of the _regular expression_ in the pattern space. Any character may be used instead of '/'. For a fuller description see _ed_(1). _Flags_ (if any) may be the following:

g Global. Substitute for all nonoverlapping instances of the _regular expression_ rather than just the first one.

p Print the pattern space if a replacement was made.

w _wfile_
Write. Append the pattern space to _wfile_ if a replacement was made.

(2)t _label_
Test. Branch to the ':' command bearing the _label_ if any substitutions have been made since the most recent reading of an input line or execution of a 't'. If _label_ is empty, branch to the end of the script.

(2)w _wfile_
Write. Append the pattern space to _wfile_.

(2)x Exchange the contents of the pattern and hold spaces.

(2)y/_string1_/_string2_/
Transform. Replace all occurrences of characters in _string1_ with the corresponding character in _string2_. The lengths of _string1_ and _string2_ must be equal.

(2)! _function_
Don't. Apply the _function_ (or group, if _function_ is '{') only to lines _not_ selected by the address(es).

(0): _label_
This command does nothing; it bears a _label_ for 'b' and 't' commands to branch to.

(1)= Place the current line number on the standard output as a line.

(2){ Execute the following commands through a matching '}' only when the pattern space is selected.

(0) An empty command is ignored.

EXAMPLE

```
sed -f sedfile inputfile >filea
```

will process the inputfile according to the _sedfile_ script, and place the results in _filea_.

The _sedfile_ script

```
4 a\
XXXXXXXXXXXXX
```

would insert a row of X's after line 4.

SEE ALSO

awk(1), ed(1), grep(1), lex(1)

McMahon, Lee E. : SED - A non-Interactive Text Editor.

NAME

see - see what a file has in it

SYNOPSIS

see [-] [name ...]

DESCRIPTION

See prints a file which contains non-printing characters in a readable format. Control characters print like ^I for tab. Delete prints as ^?. Ends of lines are marked with \`$' unless the \`-' option is given

EXAMPLE

```
see  myfile
```

displays the file *myfile* in a form like this:

```
See prints non-printing characters in a readable format.$
Control characters print like ^I for tab.$
Delete prints as ^?.$
Ends of lines are marked with `$' unless the `-' option is given$
```

where the text in the above example is a fragment of this manual page run through the *see* command.

SEE ALSO

cat(1), ex(1)

AUTHOR

Bill Joy

NAME

setmem - set user memory limit to value

SYNOPSIS

setmem [value]

DESCRIPTION

Setmem sets user memory limit to _value_ if _value_ is given. The current value is then reported. A value larger than the memory available will set the memory limit to the largest possible value.

This call is valid only on systems without memory management.

EXAMPLE

```
setmem
```

prints out the current user memory limit.

NAME

sh, for, case, if, while, :, ., break, continue, cd, eval, exec, exit, export, login, newgrp, read, readonly, set, shift, times, trap, umask, wait - command language

SYNOPSIS

sh [-ceiknrstuvx] [arg] ...

DESCRIPTION

_Sh_ is a command programming language that executes commands read from a terminal or a file. See invocation for the meaning of arguments to the shell.

Commands.
A _simple-command_ is a sequence of non blank _words_ separated by blanks (a blank is a tab or a space). The first word specifies the name of the command to be executed. Except as specified below the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see _exec_(2)). The _value_ of a simple-command is its exit status if it terminates normally or 200+_status_ if it terminates abnormally (see _signal_(2) for a list of status values).

A _pipeline_ is a sequence of one or more _commands_ separated by |. The standard output of each command but the last is connected by a _pipe_(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

A _list_ is a sequence of one or more _pipelines_ separated by ;, &, && or || and optionally terminated by ; or &. ; and & have equal precedence which is lower than that of && and ||; && and || also have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding _pipeline_ to be executed without waiting for it to finish. The symbol && (||) causes the _list_ following to be executed only if the preceding _pipeline_ returns a zero (non zero) value. Newlines may appear in a _list_, instead of semicolons, to delimit commands.

A _command_ is either a simple-command or one of the following. The value returned by a command is that of the last simple-command executed in the command.

for _name_ [in _word_ ...] do _list_ done
    Each time a for command is executed _name_ is set to the next word in the for word list If in _word_ ... is omitted then in "$@" is assumed. Execution ends when there are no more words in the list.

case _word_ in [_pattern_ [| _pattern_] ...) _list_ ;;] ... esac
    A case command executes the _list_ associated with the first pattern that matches _word_. The form of the patterns is the same as that used for file name generation.

if _list_ then _list_ [elif _list_ then _list_] ... [else _list_] fi
    The _list_ following if is executed and if it returns zero the _list_

following then is executed. Otherwise, the _list_ following elif is executed and if its value is zero the _list_ following then is executed. Failing that the else _list_ is executed.

while _list_ [do _list_] done
: A while command repeatedly executes the while _list_ and if its value is zero executes the do _list_; otherwise the loop terminates. The value returned by a while command is that of the last executed command in the do _list_. until may be used in place of while to negate the loop termination test.

(_list_)
: Execute _list_ in a subshell.

{ _list_ }
: _list_ is simply executed.

The following words are only recognized as the first word of a command and when not quoted.

    if then else elif fi case in esac for while until do done { }

Command substitution.
The standard output from a command enclosed in a pair of back quotes (``) may be used as part or all of a word; trailing newlines are removed.

Parameter substitution.
The character $ is used to introduce substitutable parameters. Positional parameters may be assigned values by set. Variables may be set by writing

    name=_value_ [_name_=_value_] ...

${_parameter_}
: A _parameter_ is a sequence of letters, digits or underscores (a _name_), a digit, or any of the characters * @ # ? - $!. The value, if any, of the parameter is substituted. The braces are required only when _parameter_ is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If _parameter_ is a digit then it is a positional parameter. If _parameter_ is * or @ then all the positional parameters, starting with $1, are substituted separated by spaces. $0 is set from argument zero when the shell is invoked.

${_parameter_-_word_}
: If _parameter_ is set then substitute its value; otherwise substitute _word_.

${_parameter_=_word_}
: If _parameter_ is not set then set it to _word_; the value of the parameter is then substituted. Positional parameters may not be

assigned to in this way.

${_parameter_?_word_}
: If _parameter_ is set then substitute its value; otherwise, print _word_ and exit from the shell. If _word_ is omitted then a standard message is printed.

${_parameter_+_word_}
: If _parameter_ is set then substitute _word_; otherwise substitute nothing.

In the above _word_ is not evaluated unless it is to be used as the substituted string. (So that, for example, echo ${d-pwd} will only execute _pwd_ if _d_ is unset.)

The following _parameters_ are automatically set by the shell.

#	The number of positional parameters in decimal.
-	Options supplied to the shell on invocation or by set.
?	The value returned by the last executed command in decimal.
$	The process number of this shell.
!	The process number of the last background command invoked.

The following _parameters_ are used but not set by the shell.

HOME	The default argument (home directory) for the cd command.
PATH	The search path for commands (see execution).
MAIL	If this variable is set to the name of a mail file then the shell informs the user of the arrival of mail in the specified file.
PS1	Primary prompt string, by default '$ '.
PS2	Secondary prompt string, by default '> '.
IFS	Internal field separators, normally space, tab, and newline.

Blank interpretation.
After parameter and command substitution, any results of substitution are scanned for internal field separator characters (those found in $IFS) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from _parameters_ that have no values) are removed.

File name generation.
Following substitution, each command word is scanned for the characters *, ? and [. If one of these characters appears then the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern then the word is left unchanged. The character . at the start of a file name or immediately following a /, and the character /, must be matched explicitly.

*	Matches any string, including the null string.
?	Matches any single character.

[...] Matches any one of the characters enclosed. A pair of characters separated by - matches any character lexically between the pair.

Quoting.
The following characters have a special meaning to the shell and cause termination of a word unless quoted.

; & ( ) | < > newline space tab

A character may be *quoted* by preceding it with a \. \newline is ignored. All characters enclosed between a pair of quote marks (´´), except a single quote, are quoted. Inside double quotes ("") parameter and command substitution occurs and \ quotes the characters \ ´ " and $.

"$*" is equivalent to "$1 $2 ..." whereas
"$@" is equivalent to "$1" "$2" ... .

Prompting.
When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a newline is typed and further input is needed to complete a command then the secondary prompt ($PS2) is issued.

Input output.
Before a command is executed its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a *command* and are not passed on to the invoked command. Substitution occurs before *word* or *digit* is used.

<*word* Use file *word* as standard input (file descriptor 0).

>*word* Use file *word* as standard output (file descriptor 1). If the file does not exist then it is created; otherwise it is truncated to zero length.

>>*word*
Use file *word* as standard output. If the file exists then output is appended (by seeking to the end); otherwise the file is created.

<<*word*
The shell input is read up to a line the same as *word*, or end of file. The resulting document becomes the standard input. If any character of *word* is quoted then no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, \newline is ignored, and \ is used to quote the characters \ $ ´ and the first character of *word*.

<&*digit*
The standard input is duplicated from file descriptor *digit*; see *dup*(2). Similarly for the standard output using >.

<&- The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit then the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example,

```
... 2>&1
```

creates file descriptor 2 to be a duplicate of file descriptor 1.

If a command is followed by & then the default standard input for the command is the empty file (/dev/null). Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input output specifications.

Environment.
The environment is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list; see _exec_(2) and _environ_(5). The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a _parameter_ for each name found, giving it the corresponding value. Executed commands inherit the same environment. If the user modifies the values of these _parameters_ or creates new ones, none of these affects the environment unless the export command is used to bind the shell's _parameter_ to the environment. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, plus any modifications or additions, all of which must be noted in export commands.

The environment for any _simple-command_ may be augmented by prefixing it with one or more assignments to _parameters_. Thus these two lines are equivalent

```
TERM=450 cmd args
(export TERM; TERM=450; cmd args)
```

If the -k flag is set, _all_ keyword arguments are placed in the environment, even if they occur after the command name. The following prints 'a=b c' and 'c':

```
echo a=b c
set -k
echo a=b c
```

Signals.
The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the shell from its parent. (But see also trap.)

Execution.
Each time a command is executed the above substitutions are carried out. Except for the 'special commands' listed below a new process is created

and an attempt is made to execute the command via an *exec*(2).

The shell parameter $PATH defines the search path for the directory containing the command. Each alternative directory name is separated by a colon (:). The default path is :/bin:/usr/bin. If the command name contains a / then the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an *a.out* file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a subshell.

Special commands.
The following commands are executed in the shell process and except where specified no input output redirection is permitted for such commands.

: No effect; the command does nothing.

. *file*
: Read and execute commands from *file* and return. The search path $PATH is used to find the directory containing *file*.

break [*n*]
: Exit from the enclosing for or while loop, if any. If *n* is specified then break *n* levels.

continue [*n*]
: Resume the next iteration of the enclosing for or while loop. If *n* is specified then resume at the *n*-th enclosing loop.

cd [*arg*]
: Change the current directory to *arg*. The shell parameter $HOME is the default *arg*.

eval [*arg* ...]
: The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...]
: The command specified by the arguments is executed in place of this shell without creating a new process. Input output arguments may appear and if no other arguments are given cause the shell input output to be modified.

exit [*n*]
: Causes a non interactive shell to exit with the exit status specified by *n*. If *n* is omitted then the exit status is that of the last command executed. (An end of file will also exit from the shell.)

export [*name* ...]
: The given names are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given then a list of exportable names is printed.

login [*arg* ...]
: Equivalent to 'exec login arg ...'.

newgrp [*arg* ...]
: Equivalent to 'exec newgrp arg ...'.

read *name* ...

One line is read from the standard input; successive words of the input are assigned to the variables *name* in order, with leftover words to the last variable. The return code is 0 unless the end-of-file is encountered.

readonly [*name* ...]
: The given names are marked readonly and the values of the these names may not be changed by subsequent assignment. If no arguments are given then a list of all readonly names is printed.

set [-eknptuvx [*arg* ...]]
- -e If non interactive then exit immediately if a command fails.
- -k All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- -n Read commands but do not execute them.
- -t Exit after reading and executing one command.
- -u Treat unset variables as an error when substituting.
- -v Print shell input lines as they are read.
- -x Print commands and their arguments as they are executed.
- \- Turn off the -x and -v options.

These flags can also be used upon invocation of the shell. The current set of flags may be found in $-.

Remaining arguments are positional parameters and are assigned, in order, to $1, $2, etc. If no arguments are given then the values of all names are printed.

shift The positional parameters from $2... are renamed $1...

times Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...
: *Arg* is a command to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. If *arg* is absent then all trap(s) *n* are reset to their original values. If *arg* is the null string then this signal is ignored by the shell and by invoked commands. If *n* is 0 then the command *arg* is executed on exit from the shell, otherwise upon receipt of signal *n* as numbered in *signal*(2). *Trap* with no arguments prints a list of commands associated with each signal number.

umask [*nnn*]
: The user file creation mask is set to the octal value *nnn* (see *umask*(2)). If *nnn* is omitted, the current value of the mask is printed.

wait [*n*]
: Wait for the specified process and report its termination status. If *n* is not given then all currently active child processes are waited for. The return code from this command is that of the

process waited for.

Invocation.
If the first character of argument zero is -, commands are read from $HOME/.profile, if such a file exists. Commands are then read as described below. The following flags are interpreted by the shell when it is invoked.

-c *string* If the -c flag is present then commands are read from *string*.

-s If the -s flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

-i If the -i flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case the terminate signal SIGTERM (see *signal*(2)) is ignored (so that 'kill 0' does not kill an interactive shell) and the interrupt signal SIGINT is caught and ignored (so that wait is interruptable). In all cases SIGQUIT is ignored by the shell.

The remaining flags and arguments are described under the set command.

FILES
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO
csh(1), test(1), exec(2),

DIAGNOSTICS
Errors detected by the shell, such as syntax errors cause the shell to return a non zero exit status. If the shell is being used non interactively then execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also exit).

BUGS
IF << is used to provide standard input to an asynchronous process invoked by &, the shell gets mixed up about naming the input document. A garbage file /tmp/sh* is created, and the shell complains about not being able to find the file by another name.

NAME

size - size of an object file

SYNOPSIS

size [-x] [object ...]

DESCRIPTION

Size prints the decimal number of bytes required by the text, data, and bss portions, and their sum in decimal, of each object-file argument. If no file is specified, a.out is used.

The -x option causes _size_ to be reported in hex.

EXAMPLE

```
size
```

prints the number of bytes for the various portions of the _a.out_ file, and their sum in decimal.

SEE ALSO

a.out(5)

NAME
sleep - suspend execution for an interval

SYNOPSIS
sleep time

DESCRIPTION
Sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time as in:

```
(sleep 105; command)&
```

or to execute a command every so often.

EXAMPLE

```
label:
     command >> x
     command >> x
     date >> x
     sleep 10
     goto label
```

would execute the two commands and append the results to file *x*, then sleep for 10 seconds, and repeat the process.

SEE ALSO
alarm(2), sleep(3)

BUGS
Time must be >0 and less than 4,294,967,295 (2**32-1) seconds, or 136 years.

NAME

sort - sort or merge files

SYNOPSIS

sort [-*mubdfinrtx*] [+*pos1* [-*pos2*]] ... [-o name] [-T directory] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '-' means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence.

The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d "Dictionary" order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option n implies option b.

r Reverse the sense of comparisons.

t*x* "Tab character" separating fields is *x*.

The notation +*pos1* -*pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. Fields are numbered starting from 0. *Pos1* and *pos2* each have the form *m*.*n*, optionally followed by one or more of the flags bdfinr, where:

m tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key.

If the b option is in effect *n* is counted from the first nonblank in the field; b is attached independently to *pos2*. A missing .*n* means the first field, .0; a missing -*pos2* means the end of the line.

Under the -t*x* option, fields are strings separated by *x*; otherwise (by default) fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. That is, if you were sorting a file whose first two fields are LastName, FirstName, the only FirstNames to be sorted (alphabetized) would be those for which the LastName was identical. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

m Merge only, the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

T The next argument is the name of a directory in which temporary files should be made.

u Unique. Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

EXAMPLES

```
sort -d +0 -1 +1 -2 addresslist
```

would sort a file of the form LastName, FirstName alphabetically by last name and by first name for last names that are identical. Two address lists that had been first sorted in this manner could then be merged with the -m option.

```
sort -t: +2n /etc/passwd
```

would print the password file (passwd(5)) sorted by user id number (the third, colon-separated, field).

FILES

/usr/tmp/stm*, /tmp/* first and second tries for temporary files

SEE ALSO

comm(1), rev(1),

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option -c.

BUGS

Very long lines are silently truncated.

NAME

spell, spellin, spellout - find spelling errors

SYNOPSIS

spell [option] ... [file] ...

spellin [list]

spellout [-d] list

DESCRIPTION

Spell collects words from the named documents, and looks them up in a spelling list. Words that do not occur in the list by fact or by derivation (by applying certain inflections, prefixes or suffices) are printed on the standard output. If no files are named, words are collected from the standard input.

Spell ignores most *troff*, *tbl* and *eqn*(1) constructions.

Under the -v option, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated.

Under the -b option, British spelling is checked. Besides preferring *centre*, *colour*, *speciality*, *travelled*, etc., this option insists upon *-ise* in words like *standardise*, (despite what Fowler and the *OED* prefer).

Under the -x option, every plausible stem is printed with '=' for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective in respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy-y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by *spell*. Both expect a list of words, one per line, from the standard input. *Spellin* adds the words on the standard input to the preexisting *list* and places a new list on the standard output. If no *list* is specified, the new list is created from scratch. *Spellout* looks up each word in the standard input and prints on the standard output those that are missing from (or present on, with option -d) the hash list.

EXAMPLE

```
spell filea fileb filec > misteaks
```

would put a list of the words in the three files that were not part of the on-line dictionary into another file, where they could be examined at leisure. The on-line dictionary rejects technical terms and proper names it does not know and treats them as equivalent to misspellings.

SEE ALSO

deroff (1), sed(1), sort(1), tee(1)

FILES

D=/usr/dict/hlist[ab]: hashed spelling lists, American & British
S=/usr/dict/hstop: hashed stop list
H=/usr/dict/spellhist: history file
/usr/lib/spell

BUGS

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions. British spelling was done by an American.

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.

EXAMPLE

```
split -100 filea newfile
```

would split filea into 100-line pieces and put them in "newfileaa", "newfilebb", and so forth until the end of filea.

NAME

ssp - make output single spaced

SYNOPSIS

ssp [name ...]

DESCRIPTION

Ssp removes extra blank lines and causes all output to be single spaced. It can be used directly, or as a filter after *nroff* or other text formatting operations.

EXAMPLE

```
nroff -ms filea fileb | ssp >> filec
```

would prepare the files with the -ms macro package, then single space the output and direct it to *filec*.

NAME

strings - find the printable strings in an object, or other binary file

SYNOPSIS

strings [-] [-o] [-_number_] file ...

DESCRIPTION

Strings looks for ascii strings in a binary file. A string is any sequence of 4 or more printing characters ending with a newline or a null. Unless the - flag is given, _strings_ only looks in the initialized data space of object files. If the -o flag is given, then each string is preceded by its offset in the file (in octal). If the -_number_ flag is given then number is used as the minimum string length rather than 4.

Strings is useful for identifying random object files and many other things.

EXAMPLE

strings objl

will locate the ASCII-character strings in the object file _objl_.

SEE ALSO

od(1)

BUGS

The algorithm for identifying strings is extremely primitive.

NAME

strip - remove symbols and relocation bits

SYNOPSIS

strip name ...

DESCRIPTION

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

The effect of *strip* is the same as use of the -s option of *ld*.

EXAMPLE

```
strip a.out
```

removes the symbol table and relocation bits from *a.out*.

FILES

/tmp/stm? temporary file

SEE ALSO

ld(1)

NAME

stty - set terminal options

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output terminal. With no argument, it reports the current settings of the options. The option strings are selected from the following set:

even	allow even parity
-even	disallow even parity
odd	allow odd parity
-odd	disallow odd parity
raw	raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
-raw	negate raw mode
cooked	same as '-raw'
cbreak	make each character available to *read*(2) as received; no erase and kill
-cbreak	make characters available to *read* only when newline is received
-nl	allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl	accept only new-line to end lines
echo	echo back every character typed
-echo	do not echo characters
lcase	map upper case to lower case
-lcase	do not map case
-tabs	replace tabs by spaces when printing
tabs	preserve tabs
ek	reset erase and kill characters back to normal # and @
erase *c*	set erase character to *c* (default control H.)
kill *c*	set kill character to *c* (default '@'.)
intr *c*	set interrupt character to *c* (default DEL.)
quit *c*	set quit character to *c* (default control \.)
start *c*	set start character to *c* (default control Q.)
stop *c*	set stop character to *c* (default control S.)
eof *c*	set end of file character to *c* (default control D.)
brk *c*	set break character to *c* (default undefined.) This character is an extra wakeup causing character.
cr0 cr1 cr2 cr3	select style of delay for carriage return (see *ioctl*(2))
nl0 nl1 nl2 nl3	select style of delay for linefeed
tab0 tab1 tab2 tab3	select style of delay for tab
ff0 ff1	select style of delay for form feed

EXAMPLE

stty

produces a list of the terminal settings currently in use. To change a setting, type in the command and the desired option. More than one option can be requested on one command line.

```
stty 300
```

sets your terminal to operate at 300 baud (hardware permitting).

```
stty >/dev/ttyl
```

reports the terminal characteristics of /dev/ttyl.

SEE ALSO
ioctl(2), tset(1), stty(2)

NAME

su - substitute user id temporarily

SYNOPSIS

su [userid]

DESCRIPTION

_Su_ demands the password of the specified _userid_, and if it is given, changes to that _userid_ and invokes the Shell _sh_(1) or _csh_(1), without changing the current directory.

The user environment is unchanged except for HOME and SHELL, which are taken from the password file for the user being substituted (see _environ_(5)). The new user ID stays in force until the Shell exits. or another _su_ is received.

If no _userid_ is specified, 'root' is assumed. Usually it is the super-user who has access to other passwords and can therefore assume other identities. To remind the super-user of his responsibilities, the Shell substitutes '#' for its usual prompt.

EXAMPLE

```
su unisoft
```

would cause the system to ask for UniSoft's password; if the password is typed in correctly, UniSoft's identity is substituted for yours, so far as the system is concerned.

SEE ALSO

csh(1), sh(1)

NAME

sum - sum and count blocks in a file

SYNOPSIS

sum file

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file, to the nearest whole block. It is typically used to look for bad spots, or to validate a file communicated over some transmission line.

EXAMPLE

```
sum sum.1
```

produces the checksum and the block count of this manual section, namely:

```
21009    1
```

SEE ALSO

wc(1)

NAME

sumdir - sum and count characters in the files in the given directories

SYNOPSIS

sumdir [directories]

DESCRIPTION

Sumdir calculates and prints a 16-bit checksum for the named file, and also prints the number of characters in the file. It is typically used to look for bad spots on the file system, or to validate a file transmitted over some transmission line. The output from this program differs from the output from the sum(1) program in that _sumdir_ prints the number of characters rather than the number of blocks in the file.

sumdir provides a recursive checksum of all files in the specified directory.

EXAMPLE

```
sumdir man1
```

produces the checksum and the character count of the files in the directory "man1".

SEE ALSO

sum(1)

NAME

sync - update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system primitive. *Sync* can be called to insure all disk writes have been completed before the processor is halted.

See *sync*(2) for details on the system primitive.

EXAMPLE

```
sync
```

should be typed to flush all internal disk buffers, before bringing down the system.

SEE ALSO

sync(2), update(1M)

NAME

tail - deliver the last part of a file

SYNOPSIS

tail ±count[lbc][r] [file]

DESCRIPTION

Tail lists the last *count* units of the specified file to the standard output. Unlike *head*, *tail* only operates on one file at a time. If no file is named, the standard input is used.

The *tail* listing can be specified to begin either + *count* units from the beginning of the file, or - *count* units from the end of the file. *Count* may be counted in units of lines, blocks or characters, according to the appended option l, b or c. When no units are specified, counting is by lines. The defail number of lines for *tail* is 10.

Specifying r causes tail to print lines from the end of the file in reverse order. The r option prints only lines starting at the specified place, and can not be combined with the [lbc] options. The default for r is to print the entire file in reverse.

EXAMPLES

```
tail +14b alpha
```

causes blocks 14 and following to be listed from the file *alpha*.

```
tail alpha
```

causes the last 10 lines to be listed from the file *alpha*.

SEE ALSO

dd(1), head(1)

BUGS

Tails selected as relative to the end of the file make use of a fixed-length buffer, and thus are limited in length.

Various kinds of anomalous behavior may happen with character special files.

NAME

take - takes a file from a remote machine.

SYNOPSIS

take [-p port] [-s[SYSID]] fromfile [tofile]

DESCRIPTION

Take takes a file onto a local machine from a remote machine. The default port is /dev/tty0; the -*p port* option can be used to specify an alternate output port. The default system id is read from /etc/sys_id, specifying generic locations for the remote machine to look for the source; the -*s*[*SYSID*] option specifies an alternate system id.

fromfile	The remote file name.
tofile	The local file name; if *tofile* is null, tofile is defaulted to fromfile. If *tofile* is ., tofile is the last component of fromfile.

NOTES

This program requires the existence of the program *takell* on the remote machine.

The -*s* option requires the existence of the file /lib/MAKE.sys on the remote machine; the option is only useful to UniSoft Systems.

SEE ALSO

put(1)

AUTHOR

UniSoft Corporation of Berkeley.

NAME

tar - tape archiver

SYNOPSIS

tar [key] [name ...]

DESCRIPTION

Tar saves and restores files. *Tar* may be used to transfer files between systems, or to save a collection of files into another file on the same system.

Tar's actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r The named files are written on the end of the tape. The c function implies this.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, the entire content of the tape is extracted. Note that if multiple entries specifying the same file are on the tape, the last one overwrites all earlier.

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

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

c Create a new tape; writing begins on the beginning of the tape instead of after the last file. This command implies r.

The following characters may be used in addition to the letter which selects the function desired.

0,...,7 This modifier selects an alternate drive on which the tape is mounted. (The default is drive 0 at 1600 bpi, which is normally /dev/rmt8.)

v Normally *tar* does its work silently. The v (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the t function, v gives more information about the tape entries than just the name.

w causes *tar* to print the action to be taken followed by file name, then wait for user confirmation. If a word beginning with 'y' is given, the action is performed. Any other input means don't do it.

f causes *tar* to use the next argument as the name of the archive instead of /dev/rmt?. If the name of the file is '-', tar writes to standard output or reads from standard input, whichever is appropriate. Thus, *tar* can be used as the head or tail of a filter chain. *Tar* can also be used to move hierarchies (see EXAMPLE).

b causes *tar* to use the next argument as the blocking factor for tape records. The default is 20, the maximum is 40. This option can be used to specify record length on raw magnetic tape archives or to cause more efficient data transfer on raw floppy disk archives. If not specified, the block size is determined automatically when reading.

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

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

Previous restrictions dealing with *tar*'s inability to properly handle blocked archives have been lifted.

EXAMPLE

```
cd fromdir; tar cf - . | (cd todir; tar xf -)
```

will copy directories from one directory tree to another.

FILES

/dev/rmt?
/tmp/tar*
/bin/mkdir build directories during recovery
/bin/pwd get working directory name

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if enough memory is not available to hold the link tables.

BUGS

There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The u option can be slow.
The current limit on file name length is 100 characters.

NAME
tbl - format tables for nroff or troff

SYNOPSIS
tbl [files] ...

DESCRIPTION
Tbl is a preprocessor for formatting tables for *nroff* or *troff*(1). 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 are reformatted. Details are given in the *tbl*(1) reference manual.

EXAMPLE
As an example, letting \t represent a tab (which should be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\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 *tbl* is used with *eqn* or *neqn* the *tbl* command should be first, to minimize the volume of data passed through pipes.

FILES

/usr/lib/tmac/tmac.s	for -ms option
/usr/lib/tmac/tmac.m	for -mm option

SEE ALSO

troff(1), eqn(1)
M. E. Lesk, TBL.

NAME

tee - pipe fitting

SYNOPSIS

tee [-i] [-a] [file] ...

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the *files*. Option -i ignores interrupts; option -a causes the output to be appended to the *files* rather than overwriting them, if the standard input is from the keyboard (not a file).

EXAMPLE

```
make | tee x
```

will cause the output of the make program to be recorded on file *x* as well as printed on standard output.

NAME

test - condition command

SYNOPSIS

test expr

DESCRIPTION

test evaluates the expression *expr*, and if its value is true then returns zero exit status; otherwise, a non zero exit status is returned. *test* returns a non zero exit if there are no arguments.

The following primitives are used to construct *expr*.

-r file true if the file exists and is readable.

-w file true if the file exists and is writable.

-f file true if the file exists and is not a directory.

-d file true if the file exists exists and is a directory.

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

-t [fildes]
true if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.

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

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

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

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

s1 true if *s1* is not the null string.

n1 -eq n2
true if the integers *n1* and *n2* are algebraically equal. Any of the comparisons -ne, -gt, -ge, -lt, or -le may be used in place of -eq.

These primaries may be combined with the following operators:

! unary negation operator

-a binary *and* operator

-o binary *or* operator

(expr)
parentheses for grouping.

-a has higher precedence than -o. Notice that all the operators and flags are separate arguments to _test_. Notice also that parentheses are meaningful to the Shell and must be escaped.

SEE ALSO

sh(1), find(1)

NAME
time - time a command

SYNOPSIS
time command

DESCRIPTION
The given command is executed; after it is complete, *time* prints the elapsed (real) time during the command, the time spent in the system, and the time spent in execution of the command.

Times are reported in seconds. The times are printed on the diagnostic output stream.

Time is also built in to *csh*(*1*), but it uses a different output format.

EXAMPLE
```
time nroff man filea
```

will, in *sh*, perform the formatting and report the time at the end of the file, e.g.:

```
real  22.0
user   8.6
sys    6.4
```

In *csh*, on the other hand, the time report might be:

```
8.9u  7.0s  0:29  54%
```

which reports the user time, system time, real time, and percentage of real time that the CPU was active, which is the sum of the user and system times divided by real elapsed time.

BUGS
Elapsed time is accurate to the second, while the CPU times are measured to your clock resolution. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

touch - update date last modified of a file

SYNOPSIS

touch [-c] file ...

DESCRIPTION

Touch attempts to set the modified date of each _file_. This is done by reading a character from the file and writing it back.

If a _file_ does not exist, an attempt will be made to create it unless the -c option is specified.

EXAMPLE

```
touch filea fileb
```

sets the "date last modified" of the two files to the current date.

SEE ALSO

utime(2)

NAME

tp - manipulate tape archive

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:

f *name* take the file "name" as the tape file name.

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 (/dev/tap?); for magtape "0" is the default (/dev/mt?).

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 cleared before beginning. Usable only with r and u. This option is assumed with magtape since it is impossible to selectively overwrite magtape.

i Errors reading and writing the tape are noted, but no action is taken. Normally, errors cause a return to the command level.

f Use the first named file, rather than a tape, as the archive. This option is known to work only with x.

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?

SEE ALSO

ar(1), tar(1)

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.

Binary-coded control information makes magnetic tapes written by _tp_ difficult to carry to other machines; _tar_(1) avoids the problem.

NAME

tr - translate characters

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*. When *string2* is short it is padded to the length of *string1* by duplicating its last character.

Any combination of the options -cds may be used:
-c complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal;
-d deletes all input characters in *string1*;
-s squeezes all strings of repeated output characters that are in *string2* to single characters.

In either string the notation *a*-*b* means a range of characters from *a* to *b* in increasing ASCII order. The character "\" followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. A "\" followed by any other character stands for that character.

EXAMPLE

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect '\' from the Shell. 012 is the ASCII code for newline.

```
tr -cs A-Za-z '\012' <file1 >file2
```

In this case, *tr* has substituted the "newline" character for all the alphabetics in *file1*, reconstituted the alphabetics with the -*c* option, squeezed the newlines to one per occurrence, with the -*s* option, and directed the output to *file2*.

SEE ALSO

ed(1), ascii(7)

BUGS

Won't handle ASCII NUL in *string1* or *string2*; always deletes NUL from input.

NAME

tra - copy out a file as it grows

SYNOPSIS

tra [-] [-interval] [+limit] file

DESCRIPTION

Tra functions similar to *cat*(1) but *tra* does not stop when it reaches the end of the file. Instead, *tra* waits for a specified interval, and if there is more information in the file, the copying process is resumed.

tra alternately copies out the new material in the file and sleeps for *interval* seconds, where the default *interval* is 15 seconds. *Limit* can be given to limit the total running time of the *tra*, the default is effectively infinite.

Tra normally copies out all the text currently in the file before beginning to watch for new text. The - option alone causes only new material to be given.

Tra is particularly useful for alternately watching the output file being written by a long shell script or a long-running program and doing real work.

AUTHOR

Bill Joy

NAME

troff, nroff - text formatting and typesetting

SYNOPSIS

troff [option] ... [file] ...

nroff [option] ... [file] ...

DESCRIPTION

Troff formats text in the named *files* for printing on a Graphic Systems C/A/T phototypesetter; *nroff* is used for for typewriter-like devices. Their capabilities are described in the *Nroff*/*Troff* *user's manual*.

If no *file* argument is present, the standard input is read. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input. The options, which may appear in any order so long as they appear before the files, are:

-o*list* Print only pages whose page numbers appear in the comma-separated *list* of numbers and ranges. A range *N*-*M* means pages *N* through *M*; an initial -*N* means from the beginning to page *N*; and a final *N*- means from *N* to the end.

-n*N* Number first generated page *N*.

-s*N* Stop every *N* pages. *Nroff* will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. *Troff* will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and resume when the typesetter's start button is pressed.

-m*name* Prepend the macro file /usr/lib/tmac/tmac.*name* to the input *files*.

-r*aN* Set register *a* (one-character) to *N*.

-i Read standard input after the input files are exhausted.

-q Invoke the simultaneous input-output mode of the rd request.

Troff only

-t Direct output to the standard output instead of the phototypesetter.

-f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-w Wait until phototypesetter is available, if currently busy.

-b Report whether the phototypesetter is busy or available. No text processing is done.

-a Send a printable ASCII approximation of the results to the standard output.

-pN Print all characters in point size _N_ while retaining all prescribed spacings and motions, to reduce phototypesetter elapsed time.

-g Prepare output for a GCOS phototypesetter and direct it to the standard output (see _gcat_(1)).

If the file /_usr_/_adm_/_tracct_ is writable, _troff_ keeps phototypesetter accounting records there. The integrity of that file may be secured by making _troff_ a 'set user-id' program.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables for _nroff_
/usr/lib/font/*	font width tables for _troff_
/dev/cat	phototypesetter
/usr/adm/tracct	accounting statistics for /dev/cat

SEE ALSO

J. F. Ossanna, _Nroff_/_Troff_ _user's_ _manual_
B. W. Kernighan, _A_ _TROFF_ _Tutorial_
eqn(1), tbl(1), ms(7), me(7), man(7)
col(1) (_nroff_ only)

NAME

true, false - provide truth values

SYNOPSIS

true

false

DESCRIPTION

True and _false_ are usually used in a Bourne shell script. They return the appropriate status "true" or "false" for running (or failing to run) a list of commands.

EXAMPLE

```
while true
do
    command list
done
```

SEE ALSO

csh(1), sh(1), false(1)

DIAGNOSTICS

True has exit status zero.

NAME

tset - set terminal modes

SYNOPSIS

tset [options]

DESCRIPTION

Tset causes terminal dependent processing such as setting erase and kill characters, setting or resetting delays, and the like. It first determines the *type* of terminal involved, names for which are specified by the /*etc*/*termcap* data base, and then does necessary initializations and mode settings. In the case where no argument types are specified, *tset* simply reads the terminal type out of the environment variable TERM and re-initializes the terminal. The rest of this manual concerns itself with type initialization, done typically once at login, and options used at initialization time to determine the terminal type and set up terminal modes.

When used in a startup script .*profile* (for *sh*(1) users) or .*login* (for *csh*(1) users) it is desirable to give information about the types of terminal usually used, for terminals which are connected to the computer through a modem. These ports are initially identified as being *dialup* or *plugboard* or *arpanet* etc. To specify what terminal type is usually used on these ports -m is followed by the appropriate port type identifier, an optional baud-rate specification, and the terminal type to be used if the mapping conditions are satisfied. If more than one mapping is specified, the first applicable mapping prevails. A missing type identifier matches all identifiers.

Baud rates are specified as with *stty*(1), and are compared with the speed of the diagnostic output (which is almost always the control terminal). The baud rate test may be any combination of: >, =, <, @, and !; @ is a synonym for = and ! inverts the sense of the test. To avoid problems with metacharacters, it is best to place the entire argument to -m within '' characters; users of *csh*(1) must also put a "\" before any "!" used here.

Thus

```
tset -m 'dialup>300:adm3a' -m dialup:dw2 -m
'plugboard:?adm3a'
```

causes the terminal type to be set to an *adm3a* if the port in use is a dialup at a speed greater than 300 baud; to a *dw2* if the port is (otherwise) a dialup (i.e. at 300 baud or less). If the *type* above begins with a question mark, the user is asked if s/he really wants that type. A null response means to use that type; otherwise, another type can be entered which will be used instead. Thus, in this case, the user will be queried on a plugboard port as to whether they are using an *adm3a*. For other ports the port type will be taken from the /etc/ttytype file or a final, default *type* option may be given on the command line not preceded by a -m.

It is often desirable to return the terminal type, as specified by the -m options, and information about the terminal to a shell's environment. This can be done using the -s option; using the Bourne shell, _sh_(1):

```
eval `tset -s options...`
```

or using the C shell, _csh_(1):

```
tset -s options ... > tset$$
source tset$$
rm tset$$
```

These commands cause _tset_ to generate as output a sequence of shell commands which place the variables TERM and TERMCAP in the environment; see _environ_(5).

Once the terminal type is known, _tset_ engages in terminal mode setting. This normally involves sending an initialization sequence to the terminal and setting the single character erase (and optionally the line-kill (full line erase)) characters.

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is the default erase character ('#' on standard systems), the erase character is changed to a Control-H (backspace).

Other options are:

-e set the erase character to be the named character _c_ on all terminals, the default being the backspace character on the terminal, usually ^H.

-k is similar to -e but for the line kill character rather than the erase character; _c_ defaults to ^X (for purely historical reasons); ^U is the preferred setting. No kill processing is done if -k is not specified.

-I supresses outputting terminal initialization strings.

-Q supresses printing the "Erase set to" and "Kill set to" messages.

-S Outputs the strings to be assigned to TERM and TERMCAP in the environment rather than commands for a shell.

EXAMPLE

A typical csh .login file using tset would be:

```
set noglob
set term = (`tset -e -S -r -d?h19`)
setenv TERM "$term[1]"
setenv TERMCAP "$term[2]"
unset term noglob
```

This .login sets the environment variables TERM and TERMCAP for the user's current terminal according to the file /etc/ttytype. If the terminal line is a dialup line, the user is prompted for the proper terminal type.

FILES

/etc/ttytype	terminal id to type map database
/etc/termcap	terminal capability database

SEE ALSO

csh(1), setenv(1), sh(1), stty(1), environ(5), ttytype(5), termcap(5)

AUTHOR

Eric Allman

BUGS

Should be merged with _stty_(1).

NOTES

For compatibility with earlier versions of _tset_ a number of flags are accepted whose use is discouraged:

-d type   equivalent to -m dialup:type

-p type   equivalent to -m plugboard:type

-a type   equivalent to -m arpanet:type

-E c   Sets the erase character to _c_ only if the terminal can backspace.

-   prints the terminal type on the standard output

-r   prints the terminal type on the diagnostic output.

NAME

tty - get terminal name

SYNOPSIS

tty

DESCRIPTION

Tty prints the pathname of the user´s terminal.

EXAMPLE

```
tty
```

produces "/dev/tty7" if user is on tty7.

DIAGNOSTICS

"Not a tty" if the standard input file is not a terminal.

NAME

ul - do underlining

SYNOPSIS

ul [-t _terminal_] [_name_ ...]

DESCRIPTION

_Ul_ reads the named files (or standard input if none are given) and translates occurences of underscores to the sequence which indicates underlining. If -t is present, _terminal_ is used as the terminal kind. Otherwise, the environment is looked in and _/etc/termcap_ read to determine the appropriate sequences for underlining. If none of the fields us, ue, or uc is present, and if so and se are present, standout mode is used to indicate underlining. If the terminal can overstrike, or handles underlining automatically, _ul_ behaves like _cat_(1). If the terminal cannot underline, underlining is ignored.

FILES

/bin/cat	concatenate and print
/etc/termcap	terminal capability data base

SEE ALSO

man(1), nroff(1)

AUTHOR

Mark Horton

BUGS

Nroff usually outputs a series of backspaces and underlines intermixed with the text to indicate underlining. No attempt is made to optimize the backward motion.

NAME

mount, umount - mount and dismount file system

SYNOPSIS

mount [special name [-r]]

umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device _special_. The file _name_ must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root. The optional argument -r indicates that the file system is to be mounted read-only.

Umount announces to the system that the removable file system previously mounted on device _special_ is to be removed.

These commands maintain a table of mounted devices in _/etc_/_mtab_. This table is only a reflection of what the mount and umount commands think is mounted, not what is actually mounted. If invoked without an argument, _mount_ prints the table.

Physically write-protected disks and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

FILES

/etc/mtab mount table

SEE ALSO

mount(2), mtab(5)

BUGS

Mounting file systems full of garbage will crash the system.
Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

NAME

unget - undo a previous get of an SCCS file

SYNOPSIS

unget [-rSID] [-s] [-n] files

DESCRIPTION

Unget undoes the effect of a get -e done prior to creating the intended new delta. If a directory is named, _unget_ behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of - is given, the standard input is read with each line being taken as the name of an SCCS file to be processed. Keyletter arguments apply independently to each named file.

-r_SID_ Uniquely identifies which delta is no longer intended. (This would have been specified by _get_ as the "new delta"). The use of this keyletter is necessary only if two or more outstanding _gets_ for editing on the same SCCS file were done by the same person (login name). A diagnostic results if the specified _SID_ is ambiguous, or if it is necessary and omitted on the command line.

-s Suppresses the printout, on the standard output, of the intended delta's _SID_.

-n Causes the retention of the gotten file which would normally be removed from the current directory.

SEE ALSO

delta(1), get(1), sact(1).

DIAGNOSTICS

Use _help_(1) for explanations.

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*(1). If the -u flag is used, just the lines that are not repeated in the original file are output. The -d option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the -u and -d mode outputs.

The -c option supersedes -u and -d and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

-*n* The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+*n* The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

sort(1), comm(1)

NAME

units - conversion program

SYNOPSIS

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
        * 2.54000e+00
        / 3.93701e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have: 15 pounds force/in2
You want: atm
        * 1.02069e+00
        / 9.79730e-01
```

Units only does multiplicative scale changes. Thus it can convert Kelvin to Rankine, but not Centigrade to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter
c	speed of light
e	charge on an electron
g	acceleration of gravity
force	same as g
mole	Avogadro's number
water	pressure head per unit height of water
au	astronomical unit

"Pound" is a unit of mass. Compound names are run together, e.g. "lightyear". British units that differ from their US counterparts are prefixed thus: "brgallon". Currency is denoted "belgiumfranc", "britainpound", ...

For a complete list of units, "cat /usr/lib/unittab".

FILES

/usr/lib/unittab complete list of units

BUGS

Don't base your financial plans on the currency conversions.

NAME

update - periodically update the super block

SYNOPSIS

update [interval].

DESCRIPTION

Update is a program that executes the *sync*(2) primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash.

If the parameter *interval* is given, it is used instead of 30 for the timing interval. This command should not be executed directly, but should be executed out of the initialization shell command file, *rc* (*8*).

SEE ALSO

sync(2), sync(1), init(1M)

NAME

updater - update files between two machines

SYNOPSIS

updater [key] local remote ...

DESCRIPTION

updater updates files between two machines.

One of the following key letters must be included:

t   Take files from the remote machine, updating the local machine.

p   Put files from the local machine onto the remote machine, updating the remote machine.

d   List the difference between files on the local and remote machines.

The following key letters are optional:

u   Update a file only if it exists on both machines; this is the default condition.

r   Replace a file if it did not exist on the destination machine.

local refers to the local directory name.

remote refers to the remote directory names. Only one remote name can be specified if the *p* (put) key is specified.

ALGORITHM

Open /dev/tty0 to the remote machine.

Stty the local port and send a stty command to the remote machine to condition both ends of the connection.

Send a "cd remote ; sumdir . | sort +2 > /tmp/rXXXXX" to remote machine for each remote system; "cd local ; sumdir . | sort > /tmp/lXXXXX" for local machine.

Wait for remote to complete.

Take /tmp/rXXXXX.

Do a comparison between the local and the union of the remotes:
    exists on remote only:
        If both the *t* and *r* keys are specified, take the file; otherwise list the file.
    exists on local only:
        If both *p* and *r* keys are specified, put the file; otherwise list the file.
    exist on both but different:

If _t_ key is specified, take the file.
If _p_ key is specified, put the file.
If _d_ key is specified, list the file.

same:
nothing

NOTES

This program is useful only to Unisoft.

AUTHOR

UniSoft Corporation of Berkeley.

NAME

uucp, uulog, uuname - unix to unix copy

SYNOPSIS

uucp [option] ... source-file ... destination-file

uulog [option] ...

uuname

DESCRIPTION

Uucp copies files named by the source-file arguments to the destination-file argument. A file name may be a path name on your machine, or may have the form

system-name!pathname

where `system-name' is taken from a list of system names which *uucp* knows about. Shell metacharacters ?*[] appearing in the pathname part will be expanded on the appropriate system.

Pathnames may be one of

(1) a full pathname;

(2) a pathname preceded by ~*user*; where *user* is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

If the result is an erroneous pathname for the remote system the copy will fail. If the destination-file is a directory, the last part of the source-file name is used.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The *uucp* command interprets the following options:

-c Use the source file when copying out rather than copying the file to the spool directory.

-d Make all necessary directories for the file copy. This is the normal action.

-e*sys* Send the *uucp* command to the system designated by *sys* to be executed there. Note that this will only be successful if the remote system allows the *uucp command to be executed there*.

-m Send mail to the requester when the copy is complete.

-r*n* indicates the role which *uucp* is to play. If *n* is 1, *uucp* acts as a master in the transaction. If *n* is 0, *uucp* acts as a slave.

-s*dir* indicates that *uucp is to use the directory dir* as the spool directory for the transfer.

EXAMPLE

uucp pascal.doc texas!~steve/pascal.doc

The *uucp* command above sends the file *pascal.doc* to the user whose name is steve, on the system called texas.

Uulog maintains a summary log of *uucp* and *uux*(1) transactions.

The options cause *uulog* to print logging information:

-s*sys* Print information about work involving system *sys*.

-u*user*
Print information about work done for the specified *user*.

The uuname utility lists the uucp names of known systems. The -l option returns the local system name.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/L.sys - List of system names and when to call them.
/usr/lib/uucp/L-dialcodes - List of 'phone numbers in L.sys.
/usr/lib/uucp/SYSTEMNAME - Name of this system.
/usr/lib/uucp/L-devices - List of device codes and speeds.
/usr/lib/uucp/USERFILE - List of users and required pathname prefixes.
/usr/lib/uucp/CMDLIST - List of commands for uuxqt to execute.
/usr/lib/uucp/uucico - copy in, copy out program; called by uucp
/usr/lib/uucp/uuxqt - command execution program; called by uucp
/usr/lib/uucp/uuclean - spool directory cleanup program; called by uucp

SEE ALSO

uux(1), mail(1)
D. A. Nowitz, Uucp Implementation Description

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will very likely not be able to fetch files by pathname; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary pathnames.

BUGS

All files received by uucp will be owned by uucp.
The -m option will only work sending files or receiving a single file. (Receiving multiple files specified by special shell characters ?*[] will not activate the -m option.)

NAME

uux - unix to unix command execution

SYNOPSIS

uux [-] command-string

DESCRIPTION

Uux will gather 0 or more files from various systems, execute a command on a specified system and send standard output to a file on a specified system.

The command-string is made up of one or more arguments that look like a shell command line, except that the command and file names may be prefixed by system-name!. A null system-name is interpreted as the local system.

File names may be one of

(1) a full pathname;

(2) a pathname preceded by ~*xxx*; where *xxx* is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

The '-' option will cause the standard input to the *uux* command to be the standard input to the command-string.

For example, the command

```
uux "!diff usg!/usr/dan/f1 pwba!/a4/dan/f1 > !fi.diff"
```

will get the f1 files from the usg and pwba machines, execute a *diff* command and put the results in f1.diff in the local directory.

Any special shell characters such as <>;| should be quoted either by quoting the entire command-string, or quoting the special characters as individual arguments.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/L.sys - List of system names and when to call them.
/usr/lib/uucp/L-dialcodes - List of 'phone numbers in L.sys.
/usr/lib/uucp/SYSTEMNAME - Name of this system.
/usr/lib/uucp/L-devices - List of device codes and speeds.
/usr/lib/uucp/USERFILE - List of users and required pathname prefixes.
/usr/lib/uucp/CMDLIST - List of commands for *uuxqt* to execute.
/usr/lib/uucp/uucico - copy in, copy out program; called by uucp
/usr/lib/uucp/uuxqt - command execution program; called by uucp
/usr/lib/uucp/uuclean - spool directory cleanup program; called by uucp

SEE ALSO

uucp(1)
D. A. Nowitz, _Uucp implementation description_

WARNING

An installation may, and for security reasons generally will, limit the list of commands executable on behalf of an incoming request from _uux_. Typically, a restricted site will permit little other than the receipt of mail via _uux_.

BUGS

Only the first command of a shell pipeline may have a system-name!. All other commands are executed on the system of the first command.
The use of the shell metacharacter * will probably not do what you want it to do.
The shell tokens << and >> are not implemented.
There is no notification of denial of execution on the remote machine.

NAME

val - validate SCCS file

SYNOPSIS

val -
val [-s] [-rSID] [-mname] [-ytype] files

DESCRIPTION

Val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of keyletter arguments, which begin with a -, and named files.

Val has a special argument, -, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

Val generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

The keyletter arguments are defined as follows. The effects of any keyletter argument apply independently to each named file on the command line.

-s
: The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

-r*SID*
: The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., r1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc. which may exist) or invalid (for example, r1.0 or r1.1.0 are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

-m*name*
: The argument value *name* is compared with the SCCS %M% keyword in *file*.

-y*type*
: The argument value *type* is compared with the SCCS %Y% keyword in *file*.

The 8-bit code returned by *val* is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;

bit 2 = corrupted SCCS file;
bit 3 = can't open file or file not SCCS;
bit 4 = _SID_ is invalid or ambiguous;
bit 5 = _SID_ does not exist;
bit 6 = %Y%, -y mismatch;
bit 7 = %M%, -m mismatch;

Note that _val_ can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned - a logical OR of the codes generated for each command line and file processed.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

DIAGNOSTICS

Use _help_(1) for explanations.

BUGS

Val can process up to 50 files on a single command line. Any number above 50 will produce a core dump.

NAME

vc - version control

SYNOPSIS

vc [-a] [-t] [-cchar] [-s] [keyword=value ... keyword=value]

DESCRIPTION

The _vc_ command copies lines from the standard input to the standard output under control of its _arguments_ and _control statements_ encountered in the standard input. In the process of performing the copy operation, user declared _keywords_ may be replaced by their string _value_ when they appear in plain text and/or control statements.

The copying of lines from the standard input to the standard output is conditional, based on tests (in control statements) of keyword values specified in control statements or as _vc_ command arguments.

A control statement is a single line beginning with a control character, except as modified by the -t keyletter (see below). The default control character is colon (:), except as modified by the -c keyletter (see below). Input lines beginning with a backslash (\) followed by a control character are not control lines and are copied to the standard output with the backslash removed. Lines beginning with a backslash followed by a non-control character are copied in their entirety.

A keyword is composed of 9 or less alphanumerics; the first must be alphabetic. A value is any ASCII string that can be created with _ed_(1); a numeric value is an unsigned string of digits. Keyword values may not contain blanks or tabs.

Replacement of keywords by values is done whenever a keyword surrounded by control characters is encountered on a version control statement. The -a keyletter (see below) forces replacement of keywords in _all_ lines of text. An uninterpreted control character may be included in a value by preceding it with \. If a literal \ is desired, then it too must be preceded by \.

Keyletter arguments

-a
: Forces replacement of keywords surrounded by control characters with their assigned value in _all_ text lines and not just in _vc_ statements.

-t
: All characters from the beginning of a line up to and including the first _tab_ character are ignored for the purpose of detecting a control statement. If one is found, all characters up to and including the _tab_ are discarded.

-c_char_
: Specifies a control character to be used in place of :.

-s Silences warning messages (not error) that are normally printed on the diagnostic output.

Version Control Statements

:dcl keyword[, ..., keyword]
Used to declare keywords. All keywords must be declared.

:asg keyword=value
Used to assign values to keywords. An asg statement overrides the assignment for the corresponding keyword on the _vc_ command line and all previous asg´s for that keyword. Keywords declared, but not assigned values have null values.

:if condition
.
.
.
:end
Used to skip lines of the standard input. If the condition is true all lines between the _if_ statement and the matching _end_ statement are copied to the standard output. If the condition is false, all intervening lines are discarded, including control statements. Note that intervening _if_ statements and matching _end_ statements are recognized solely for the purpose of maintaining the proper _if_-_end_ matching.
The syntax of a condition is:

```
<cond>  ::= [ "not" ] <or>
<or>    ::= <and> | <and> "|" <or>
<and>   ::= <exp> | <exp> "&" <and>
<exp>   ::= "(" <or> ")" | <value> <op> <value>
<op>    ::= "=" | "!=" | "<" | ">"
<value> ::= <arbitrary ASCII string> | <numeric string>
```

The available operators and their meanings are:

=	equal
!=	not equal
&	and
|	or
>	greater than
<	less than
( )	used for logical groupings
not	may only occur immediately after the _if_, and when present, inverts the value of the entire condition

The > and < operate only on unsigned integer values (e. g.: 012 > 12 is false). All other operators take strings as arguments (e. g.: 012 != 12 is true). The precedence of the operators (from highest to lowest) is:

```
= != > <     all of equal precedence
&
|
```

Parentheses may be used to alter the order of precedence.
Values must be separated from operators or parentheses by at least one blank or tab.

::text
Used for keyword replacement on lines that are copied to the standard output. The two leading control characters are removed, and keywords surrounded by control characters in text are replaced by their value before the line is copied to the output file. This action is independent of the -a keyletter.

:on

:off
Turn on or off keyword replacement on all lines.

:ctl char
Change the control character to char.

:msg message
Prints the given message on the diagnostic output.

:err message
Prints the given message followed by:

    ERROR: err statement on line ... (915)

on the diagnostic output. Vc halts execution, and returns an exit code of 1.

DIAGNOSTICS

Use help(1) for explanations.

EXIT CODES

0 - normal
1 - any error

NAME

version - reports version number of files

SYNOPSIS

version name ...

DESCRIPTION

Version takes a list of files and reports the version number. If the file is not a binary, it reports: "not a binary". If no version number is associated with the file, it reports: "pre history". Version is useful for determining which version of the current program you are running.

EXAMPLE

```
version /bin/version
```

prints the version number of the version program.

NAME

vi - screen oriented (visual) display editor based on ex

SYNOPSIS

vi [-t tag] [-r] [+_command_] [-w_n_] name ...

DESCRIPTION

_Vi_ (visual) is a display oriented text editor based on _ex_(1). _Ex_ and _vi_ run the same code; it is possible to get to the command mode of _ex_ from within _vi_ and vice-versa.

_Vi_ puts up a screenful of text at a time (unless a smaller window is specified) and allows rapid and fluid cursor motion to the place where you want to begin adding, changing, or deleting text. With _vi_, editing can be done on characters, words, lines, or sections at a time. When multi-character changes are made, it is necessary to hit the ESCAPE key to return to cursor motion mode.

Using _ex_ commands and calling up the Shell by typing (!) are done with a colon (:) and the appropriate command sequence, such as that to find a string or write the file.

The "Vi Command Summary" (below), the _Vi Quick Reference_ card and the _Introduction to Display Editing with Vi_ provide full details on using _vi_.

The following options are recognized:

-t Equivalent to an initial _tag_ command, editing the file containing the _tag_ and positioning the editor at its definition.

-r Used in recovering after an editor or system crash, retrieving the last saved version of the named file. If no file is specified, a list of saved files will be reported.

+_command_
indicates that the editor should begin by executing the specified command. If _command_ is omitted, then it defaults to "$", positioning the editor at the last line of the first file initially. Other useful commands here are scanning patterns of the form "/pat" or line numbers, e.g. "+100" to start at line 100.

-w_n_ sets the default window size to _n_, and is useful in dialups, to start in small windows.

Name arguments indicate files to be edited.

VI COMMAND SUMMARY

Cursor Motion:	Forward	Back
letter	(space)	^H, h
word right-limit	E,e	
word left-limit	W,w	B,b
sentence	)	(
paragraph	}	{
section/function	]]	[[
line: same/limit	$	0
1st charac	+,<ret>	-
same column	^n,LF	^p
specified	<line#>G	<line#>G
1/2 screenful	^d	^u
screenful	^f	^b

Undoing Errors

(see also: change, insert, delete)

u	undo last change
U	restore current line
"Np	retrieve Nth last delete
<esc>	abandon incomplete command (without completing it)
:q!	drastic! abandon without saving.

Insert	Change	Delete
i before cursor	cw<newword> word	x character
I before 1st non-blank	C substitute line	X ....before cursor
a after cursor	s substitute charac.	dw word
A at end-of-line	S subst. lines	de ....but leave punctuation
o open line below	rx replace 1 charac	dd line
O open line above	R replace characs	(#)dd number of lines
<esc> terminates insert	xp transpose charac	D rest of line
	<esc> terminates change	

Delete during Insert

last charac	^H
last word	^W
all input this line	<@>

FILES

See *ex*(1).

SEE ALSO

ex (1), edit (1), "Vi Quick Reference" card, "An Introduction to Display Editing with Vi".

AUTHOR

William Joy
Mark Horton added macros to _visual_ mode and is maintaining version 3

BUGS

Software tabs using ^T work only immediately after the _autoindent_.

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The _wrapmargin_ option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Saving text on deletes in the named buffers is somewhat inefficient.

The _source_ command does not work when executed as :source; there is no way to use the :append, :change, and :insert commands, since it is not possible to give more than one line of input to a : escape. To use these on a :global you must Q to _ex_ command mode, execute them, and then reenter the screen editor with _vi_ or _open_.

NAME
wait - await completion of process

SYNOPSIS
wait

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

Because the _wait_(2) system call must be executed in the parent process, the Shell itself executes _wait_, without creating a new process.

EXAMPLE
```
wait
```

waits for all child processes to terminate.

SEE ALSO
sh(1)

BUGS
Not all the processes of a 3- or more-stage pipeline are children of the Shell, and thus can't be waited for. (This bug does not apply to _csh_(1).)

NAME

wall - write to all users

SYNOPSIS

wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends the message, preceded by "Broadcast Message ...", to all logged in users.

Only the super-user can override any protections against receiving messages that users may have invoked. The message is also labeled with the sender's name and terminal number and the time the message was sent.

EXAMPLE

```
wall
```

will broadcast the standard input to all users who are not protected against receiving messages by the _mesg_ command.

FILES

/dev/tty?
/etc/utmp

SEE ALSO

mesg(1), write(1)

DIAGNOSTICS

"Cannot send to ..." when the open on a user's tty file fails.

NAME

wc - word count

SYNOPSIS

wc [-lwc] [name ...]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of characters delimited by spaces, tabs or newlines.

If an argument beginning with one of "lwc" is present, the specified counts (lines, words, or characters) are selected by the letters l, w, or c. Note that the default options are: -lwc.

EXAMPLE

```
wc filea fileb filec
```

reports the number of lines, words, and characters in each of the files.

NAME

what - identify SCCS files

SYNOPSIS

what files

DESCRIPTION

What searches the given files for all occurrences of the pattern that *get*(1) substitutes for %Z% (this is @(#) at this printing) and prints out what follows until the first ", >, new-line, \, or null character. For example, if the C program in file f.c contains

```
char ident[] = "@(#)identification information";
```

and f.c is compiled to yield f.o and a.out, then the command

```
what f.c f.o a.out
```

will print

```
f.c:
        identification information

f.o:
        identification information

a.out:
        identification information
```

What is intended to be used in conjunction with the SCCS command *get*(1), which automatically inserts identifying information, but it can also be used where the information is inserted manually.

SEE ALSO

get(1), help(1).

DIAGNOSTICS

Use *help*(1) for explanations.

BUGS

It's possible that an unintended occurrence of the pattern @(#) could be found just by chance, but this causes no harm in nearly all cases.

NAME

whereis - locate source/binary/manual for program

SYNOPSIS

whereis [-sbmu] [-SBM dir ... [-f]] name ...

DESCRIPTION

Whereis locates source, binary and manual sections for specified files. The supplied names are first stripped of leading pathname components and any (single) trailing extension of the form ".ext", e.g. ".c". Prefixes of "s." resulting from use of source code control are also dealt with. *Whereis* then attempts to locate the desired program in a list of standard places. If any of the -b, -s or -m flags are given then *whereis* searches only for binaries, sources or manual sections (or any two thereof).

The -u flag may be used to search for unusual entries. A file is said to be unusual if it does not have one entry of each requested type. Thus "whereis -m -u *" asks for those files in the current directory which have no documentation.

Finally, the -B -M and -S flags may be used to change the places where *whereis* searches to the specified directories only. The -f file flags may be used to terminate the last such directory list and signal the start of file names.

EXAMPLE

The following finds all the files in /usr/ucb which are not documented in /usr/man/mann with source in /usr/ucb/src/ucb:

```
cd /usr/ucb
whereis -u -M /usr/man/mann -S /usr/ucb/src/ucb -f *
```

FILES

/usr/src/*
/usr/man/*
/bin
/etc
/usr/bin
/usr/games
/lib
/usr/lib

AUTHOR

Bill Joy

DIAGNOSTICS

None.

BUGS

This program makes it too easy to find out what needs to be done.

Since the program uses _chdir_(1) to run faster, pathnames given with the -M -S and -B flags should start at the root or they will not work.

NAME

who - who is on the system

SYNOPSIS

who [who-file] [am I]

DESCRIPTION

Who, without an argument, lists the login name, terminal name, and login time for each current UNIX user.

Without an argument, *who* examines the /etc/utmp file to obtain its information. If a file is given, that file is examined. Typically the given file will be /usr/adm/wtmp, which contains a record of all the logins since it was created. Then *who* lists logins, logouts, and crashes since the creation of the wtmp file. Each login is listed with user name, terminal name (with '/dev/' suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with 'x' in the place of the device name, and a fossil time indicative of when the system went down.

With two arguments, as in 'who am I' (and also 'who are you'), *who* tells who you are logged in as.

EXAMPLE

```
who am i
```

reports the name under which you are currently logged in. This could be a name other than the original name under which you logged in, if the *su* command has been used.

FILES

/etc/utmp

SEE ALSO

getuid(2), su(1), utmp(5)

NAME

whoami - print effective current user id

SYNOPSIS

whoami

DESCRIPTION

Whoami prints who you are, the name you logged in under originally. It works even if you are using a substitute ID with _su_, while 'who am i' does not, since it uses /etc/utmp.

EXAMPLE

```
whoami
```

might reply:

```
unisoft
```

FILES

/etc/passwd	User data base
/etc/utmp	login records

SEE ALSO

who (1)

NAME

write - write to another user

SYNOPSIS

write user [ttyname]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message

Message from yourname yourttyname...

The recipient of the message should write back at this point. Communication continues until an end of file (Control-d) is read from the terminal or an interrupt is sent. At that point *write* writes 'EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *ttyname* argument may be used to indicate the appropriate terminal name.

Permission to write may be denied or granted by use of the *mesg* command. At the outset writing is allowed. Certain commands, in particular *nroff* and *pr*(1) disallow messages in order to prevent messy output.

If the character '!' is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first write to another user, wait for him to write back before starting to send. Each party should end each message with a distinctive signal: (o) for 'over' is conventional. This signals the other for a reply. (oo) for 'over and out' is suggested when conversation is about to be terminated with a Control-d.

EXAMPLE

write unisoft tty7

writes unisoft on terminal 7, unless messages have been refused with *mesg*(*1*).

FILES

/etc/utmp	to find user
/bin/sh	to execute '!'

SEE ALSO

mail(1), mesg(1), who(1)

NAME

xstr - extract strings from C programs to implement shared strings

SYNOPSIS

xstr [-c] [-] [file]

DESCRIPTION

Xstr maintains a file *strings* into which strings in component parts of a large program are hashed. The strings in the programs modules are replaced with pointers to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

```
xstr -c name
```

will extract the strings from the C source in the file *name*, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file.

The resulting C text is placed in the file *x.c*, after which it can be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffixes of existing strings do not cause changes to the data base.

After all components of a large program have been compiled a file *xs.c* declaring the common *xstr* space can be created by a command of the form

```
xstr
```

This *xs.c* file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. A command

```
xstr name
```

creates files *x.c* and *xs.c* as before, without using or affecting any other *strings* or C text file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *Xstr* reads from its standard input when the argument \`-' is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

```
cc -E name.c | xstr -c -
cc -c x.c
mv x.o name.o
```

Xstr does not touch the file *strings* unless new items are added, thus *make* can avoid remaking *xs.o* unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array `xstr'
/tmp/xs*	Temp file when `xstr name' doesn't touch *strings*

SEE ALSO

mkstr(1)

AUTHOR

Bill Joy

BUGS

If a string is a suffix of another string in the data base, but the shorter string is seen first by *xstr* both strings will be placed in the data base, when just placing the longer one there will do.

NAME

yacc - yet another compiler-compiler

SYNOPSIS

yacc [-vd] grammar

DESCRIPTION

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, *y.tab.c*, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *Lex*(1) is useful for creating lexical analyzers usable by *yacc*.

If the -v flag is given, the file *y.output* is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the -d flag is used, the file *y.tab.h* is generated with the *define* statements that associate the *yacc*-assigned 'token codes' with the user-declared 'token names'. This allows source files other than *y.tab.c* to access the token codes.

FILES

y.output	
y.tab.c	
y.tab.h	defines for token names
yacc.tmp, yacc.acts	temporary files
/usr/lib/yaccpar	parser prototype for C programs

SEE ALSO

lex(1)
LR Parsing by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974.
YACC - Yet Another Compiler Compiler by S. C. Johnson.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the *y.output* file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

BUGS

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

NAME

intro, errno - introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

Section 2 of this manual describes all the entries into the system. Distinctions as to the status of the entries are made in the headings:

(2) System call entries which are standard in Version 7 UNIX systems.

An error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details. An error number is also made available in the external variable _errno_. _Errno_ is 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_(3). The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in <errno.h>, and the messages available using _perror_(3).

0 Error 0
Unused.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
The process whose number was given to _signal_ and _ptrace_ does not exist, or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error occurred during a _read_ or _write_. This error may in some cases occur on a call following the one to which

it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialed in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5120 bytes is presented to _exec_.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see _a.out_(5).

9 EBADF Bad file number
Either a file descriptor refers to no open file, 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 or the user is not allowed to create any more processes. This error may also occur when there is not enough swap space to hold a process.

12 ENOMEM Not enough core
During an _exec_ or _break_, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g. in _mount_.

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file directory. (open file, current directory, mounted-on file,

active text segment).

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g. _link_.

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to _chdir_.

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in _signal_, reading or writing a file for which _seek_ has generated a negative pointer. Also set by math functions, see _intro_(3).

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more _opens_ can be accepted.

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in _stty_ or _gtty_ is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure shared text 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
The size of a file exceeded the maximum (about 1.0E9 bytes).

28 ENOSPC No space left on device
During a _write_ to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An _lseek_ was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 links to a file.

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

35 EDEADLOCK Locking deadlock
Returned by locking(2) system call if deadlock would occur or when locktable overflows.

SEE ALSO

intro(3)

BUGS

The message Mount device busy is reported when a terminal is inaccessible because the exclusive use bit is set; this is confusing.

NAME

access - determine accessibility of file

SYNOPSIS

```
access(name, mode)
char *name;
int mode;
```

DESCRIPTION

Access checks the given file _name_ for accessibility according to _mode_, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if _name_ cannot be found or if any of the desired access modes would not be granted. On disallowed accesses -1 is returned and the error code is in _errno_. 0 is returned from successful tests.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by _access_, but an attempt to open it for writing will fail because it is not allowed to write into the directory structure itself, although files may be created there. A file may look executable, but _exec_ will fail unless it is in proper format.

SEE ALSO

stat(2)

ASSEMBLER

```
movl #33,D0
movl #name,A0
movl mode,D1
trap #0
```

Carry bit cleared on success.

NAME

acct - turn accounting on or off

SYNOPSIS

```
acct(file)
char *file;
```

DESCRIPTION

The system is prepared to write a record in an accounting *file* for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to *file*. An argument of 0 causes accounting to be turned off.

The accounting file format is given in *acct*(5).

SEE ALSO

acct(5)

DIAGNOSTICS

On error -1 is returned. The file must exist and the call may be exercised only by the super-user. It is erroneous to try to turn on accounting when it is already on.

BUGS

No accounting is produced for programs running when a crash occurs. In particular, nonterminating programs are never accounted for.

ASSEMBLER

```
movl #51,D0
movl #file,A0
trap #0
```

Carry bit cleared on success.

NAME

alarm - schedule signal after specified time

SYNOPSIS

```
alarm(seconds)
unsigned seconds;
```

DESCRIPTION

Alarm causes signal SIGALRM, see *signal*(2), to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is canceled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 4,294,967,295 (2**32-1) seconds, or 136 years.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), signal(2), sleep(3)

ASSEMBLER

```
movl #27,D0
movl seconds,A0
trap #0
```

D0 will contain the amount of time previously remaining in the alarm clock.

NAME

brk, sbrk, break - change core allocation

SYNOPSIS

```
char *brk(addr)
char *addr;

char *sbrk(incr)
int incr;
```

DESCRIPTION

Brk sets the system's idea of the lowest location not used by the program (called the break) to *addr* rounded up to the next memory segment multiple. Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In systems without memory management *brk* will fail if there are not at least 8192 bytes between the top of the permanent data space and the bottom of the current stack pointer.

In the alternate function *sbrk*, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via *exec*, the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *break*.

SEE ALSO

exec(2), malloc(3), end(3)

DIAGNOSTICS

On success *brk* and *sbrk* return pointers to the beginning of the new area; -1 is returned if the program requests more memory than the system limit or, on memory management CPUs, if too many segmentation registers would be required to implement the break. *Sbrk* returns -1 if the break could not be set.

ASSEMBLER

```
movl #17,D0
movl #addr,A0
trap #0
```

Carry bit cleared if the *brk* could be set; *brk* fails if the program requests more memory than the system limit or, on memory management CPUs, if too many segmentation registers would be required to implement the break.

NAME
chdir - change current working directory

SYNOPSIS
```
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
cd(1)

DIAGNOSTICS
Zero is returned if the directory is changed; -1 is returned if the given name is not that of a directory or is not searchable by the user.

ASSEMBLER
```
movl #12,D0
movl #dirname,A0
trap #0
```

Carry bit cleared on success.

NAME

chmod - change mode of file

SYNOPSIS

```
chmod(name, mode)
char *name;
int mode;
```

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by _name_ has its mode changed to _mode_. Modes are constructed by _or_ing together some combination of the following:

04000 set user ID on execution

02000 set group ID on execution

01000 save text image after execution (for shareable files)

00400 read by owner

00200 write by owner

00100 execute (search on directory) by owner

00070 read, write, execute (search) by group

00007 read, write, execute (search) by others

If an executable file is set up for sharing (see the _cc_ -n option), 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. Ability to set this bit is restricted to the super-user since swap space is consumed by the images.

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

Changing the owner of a file turns off the set-user-id bit. This makes the system somewhat more secure by protecting set-user-id files from remaining set-user-id if they are modified, at the expense of a degree of compatibility.

SEE ALSO

chmod(1)

DIAGNOSTIC

Zero is returned if the mode is changed; -1 is returned if _name_ cannot be found or if the current user is neither the owner of the file nor the super-user.

ASSEMBLER

```
movl #15,D0
movl #name,A0
movl mode,D1
trap #0
```

Carry bit cleared on success.

NAME

chown - change owner and group of a file

SYNOPSIS

```
chown(name, owner, group)
char *name;
int owner;
int group;
```

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by _name_ has its _owner_ and _group_ changed as specified. Only the super-user may execute this call.

Chown clears the set-user-id bit on the file to prevent accidental creation of set-user-id programs owned by the super-user.

SEE ALSO

chown(1), passwd(5)

DIAGNOSTICS

Zero is returned if the owner is changed; -1 is returned on illegal owner changes.

ASSEMBLER

```
movl #16,D0
movl #name,A0
movl owner,D1
movl group,A1
trap #0
```

Carry bit cleared on success.

NAME

close - close a file

SYNOPSIS

```
close(fildes)
int fildes;
```

DESCRIPTION

Given a file descriptor such as returned from an _open_, _creat_, _dup_ or _pipe_(2) call, _close_ closes the associated file. A close of all files is automatic on _exit_, but since there is a 20 open file limit on the number of open files per process, _close_ is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain high-numbered file descriptors are closed by _exec_(2), and it is possible to arrange for others to be closed (see FIOCLEX in _ioctl_(2)).

SEE ALSO

creat(2), open(2), pipe(2), exec(2), ioctl(2)

DIAGNOSTICS

Zero is returned if a file is closed; -1 is returned for an unknown file descriptor.

ASSEMBLER

```
movl #6,D0
movl fildes,A0
trap #0
```

Carry bit cleared on success.

NAME

creat - create a new file

SYNOPSIS

```
creat(name, mode)
char *name;
int mode;
```

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask (see *umask*(2)). Also see *chmod*(2) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is opened for writing only (not reading), and its file descriptor is returned.

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat*, an error is returned and the program knows that the name is unusable for the moment.

The system scheduling algorithm does not make this a true uninterruptible operation, and a race condition may develop if *creat* is done at precisely the same time by two different processes.

SEE ALSO

write(2), close(2), chmod(2), umask (2)

DIAGNOSTICS

The value -1 is returned if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is directory; there are already too many files open.

ASSEMBLER

```
movl #8,D0
movl #name,A0
movl mode,D1
trap #0
```

Carry bit cleared on success.

The file descriptor is returned in D0.

NAME

dup, dup2 - duplicate an open file descriptor

SYNOPSIS

```
dup(fildes)
int fildes;

dup2(fildes, fildes2)
int fildes;
int fildes2;
```

DESCRIPTION

Given a file descriptor returned from an _open_, _pipe_, or _creat_ call, _dup_ allocates another file descriptor synonymous with the original. The new file descriptor is returned.

In the second form of the call, _fildes_ is a file descriptor referring to an open file, and _fildes2_ is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). _Dup2_ causes _fildes2_ to refer to the same file as _fildes_. If _fildes2_ already referred to an open file, it is closed first.

SEE ALSO

creat(2), open(2), close(2), pipe(2)

DIAGNOSTICS

The value -1 is returned if: the given file descriptor is invalid; there are already too many open files.

ASSEMBLER

```
movl #41,D0
movl fildes,A0
trap #0
```

Carry bit cleared on success.

The _dup2_ entry is implemented by adding 0100 to _fildes_.

NAME

execl, execv, execle, execve, execlp, execvp, exec, exece, environ - execute a file

SYNOPSIS

```
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[];

execle(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

execve(name, argv, envp)
char *name, *argv[], *envp[];

extern char **environ;
```

DESCRIPTION

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

Files remain open across *exec* unless explicit arrangement has been made; see *ioctl*(2). Ignored/held signals remain ignored/held across these calls, but signals that are caught (see *signal*(2)) are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. *Exec* changes the effective user and group ID to the owner of the executed file if the file has the 'set-user-ID' or 'set-group-ID' modes. The real user ID is not affected.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

From C, two interfaces are available. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments; the first argument is conventionally the same as the 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, envp)
int argc;
char **argv, **envp;
```

where _argc_ is the argument count and _argv_ is an array of character pointers to the arguments themselves. As indicated, _argc_ is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is directly usable in another _execv_ because _argv_[_argc_] is 0.

Envp is a pointer to an array of strings that constitute the _environment_ of the process. Each string consists of a name, an =, and a null-terminated value. The array of pointers is terminated by a null pointer. The shell _sh_(1) passes an environment entry for each global shell variable defined when the program is called. See _environ_(5) for some conventionally used names. The C run-time start-off routine places a copy of _envp_ in the global cell _environ_, which is used by _execv_ and _execl_ to pass the environment to any subprograms executed by the current program. The _exec_ routines use lower-level routines as follows to pass an environment explicitly:

```
execve(file, argv, environ);
execle(file, arg0, arg1, . . . , argn, 0, environ);
```

Execlp and _execvp_ are called with the same arguments as _execl_ and _execv_, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

FILES

/bin/sh shell, invoked if command file found by _execlp_ or _execvp_

SEE ALSO

fork(2), environ(5), csh(1)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see _a.out_(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is -1. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

BUGS

If _execvp_ is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of _argv_[_0_] and _argv_[_-1_] will be modified before return.

ASSEMBLER

```
movl #11,D0                    | sys exec
```

```
movl #name,A0
movl #argv,D1
trap #0

movl #59                            | sys exece
movl #name,A0
movl #argv,D1
movl #envp,A1
trap #0
```

Plain _exec_ is obsoleted by _exece_, but remains for historical reasons.

When the called file starts execution, 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, followed by a null pointer, followed by the pointers to the environment strings and then another null pointer. The strings themselves follow; a 0 word is left at the very top of memory.

```
               nargs                | stack points here
               arg0
               ...
               argn
               0
               env0
               ...
               envm
               0

arg0:          <arg0\0>
               ...
env0:          <env0\0>
               0
```

NAME

exit - terminate process

SYNOPSIS

```
exit(status)
int status;

_exit(status)
int status;
```

DESCRIPTION

Exit is the normal means of terminating a process. *Exit* closes all the process's files and notifies the parent process if it is executing a *wait*. The low-order 8 bits of *status* are available to the parent process.

This call can never return.

The C function *exit* may cause cleanup actions before the final 'sys exit'. The function "_exit" circumvents all cleanup, and should be used to terminate a child process after a *fork*(2) to avoid flushing buffered output twice.

SEE ALSO

fork(2), wait(2)

ASSEMBLER

```
movl #1,D0
movl status,A0
trap #0
```

NAME

fork - spawn new process

SYNOPSIS

fork()

DESCRIPTION

Fork is the only way a new process is created. With _fork_, the new process's core image is a copy of that of the caller of _fork_. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID can be used when doing a _wait_(2).

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

SEE ALSO

wait(2), exec(2)

DIAGNOSTICS

Returns -1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full.

ASSEMBLER

```
movl #2,D0
trap #0
```

Carry bit cleared on success.

New process return.
Old process return, new process ID in D0.

The return locations in the old and new process differ by one 16 bit word. The C-bit is set in the old process if a new process could not be created.

NAME

getpid - get process identification

SYNOPSIS

```
getpid()
```

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

SEE ALSO

mktemp(3)

ASSEMBLER

```
movl #20,D0
trap #0
```

Process ID is returned in D0.

NAME

getuid, getgid, geteuid, getegid - get user and group identity

SYNOPSIS

getuid()

geteuid()

getgid()

getegid()

DESCRIPTION

Getuid returns the real user ID of the current process, *geteuid* the effective user ID. The real user ID identifies the person who is logged in, in contrast to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the 'set user ID' mode, to find out who invoked them.

Getgid returns the real group ID, *getegid* the effective group ID.

SEE ALSO

setuid(2)

ASSEMBLER

```
movl #24,D0        | sys getuid
trap #0
```

Real user ID in D0, effective user ID in D1.

```
movl #47,D0        | sys getgid
trap #0
```

Real group ID in D0, effective group ID in D1.

NAME

ioctl, stty, gtty - control device

SYNOPSIS

```
#include <sgtty.h>

ioctl(fildes, request, argp)
int fildes;
int request;
struct sgttyb *argp;

stty(fildes, argp)
int fildes;
struct sgttyb *argp;

gtty(fildes, argp)
int fildes;
struct sgttyb *argp;
```

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in section 4 discuss how *ioctl* applies to them.

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *tty*(4).

The following two standard calls, however, apply to any open file:

```
ioctl(fildes, FIOCLEX, NULL);
ioctl(fildes, FIONCLEX, NULL);
```

The first causes the file to be closed automatically during a successful *exec* operation; the second reverses the effect of the first.

The following call applies to any open file:

```
ioctl(fildes, FIONREAD, &count)
```

returning, in the longword *count* the number of characters available for reading from *fildes*.

SEE ALSO

stty(1), tty(4), exec(2)

DIAGNOSTICS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended, or if _request_ attempts to modify the state of a terminal when _fildes_ is not writeable.

BUGS

Strictly speaking, since _ioctl_ may be extended in different ways to devices with different properties, _argp_ should have an open-ended declaration like

```
union { struct sgttyb ...; ... } *argp;
```

The important thing is that the size is fixed by 'struct sgttyb'.

ASSEMBLER

```
movl #54,D0          | sys ioctl
movl fildes,A0
movl request,D1
movl #argp,A1
trap #0
```

Carry bit cleared on success.

```
movl #31,D0          | sys stty
movl fildes,A0
movl #argp,D1
trap #0
```

Carry bit cleared on success.

```
movl #32,D0          | sys gtty
movl fildes,A0
movl #argp,D1
trap #0
```

Carry bit cleared on success.

NAME

kill - send signal to a process

SYNOPSIS

```
kill(pid, sig)
int pid;
int sig;
```

DESCRIPTION

Kill sends the signal _sig_ to the process specified by the process number _pid_. See _signal_(2) for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all processes in the sender's process group; see _tty_(4).

If the process number is -1, and the user is the super-user, the signal is broadcast universally except to processes 0, 1, the scheduler initialization, and the process sending the signal.

Processes may send signals to themselves.

SEE ALSO

signal(2), kill(1), init(1M)

DIAGNOSTICS

Zero is returned if the process is killed; -1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

ASSEMBLER

```
movl #37,D0
movl pid,A0
movl sig,D1
trap #0
```

Carry bit cleared on success.

NAME

link - link to a file

SYNOPSIS

```
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. The linked file is actually a pointer to the original file. When the last link to a file is removed the file is deleted.

SEE ALSO

ln(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; -1 is returned when name1 cannot be found; when name2 already exists; when the directory of name2 cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when a file has more than 32767 links.

On some systems the super-user may link to non-ordinary files.

ASSEMBLER

```
movl #9,D0
movl #name1,A0
movl #name2,D1
trap #0
```

Carry bit cleared on success.

NAME

lock - lock a process in primary memory

SYNOPSIS

```
lock(flag)
int flag;
```

DESCRIPTION

If the _flag_ argument is non-zero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is un_lock_ed. This call may only be executed by the super-user.

BUGS

_Lock_ed processes interfere with the compaction of primary memory and can cause a system deadlock.

ASSEMBLER

```
movl #53,D0
movl flag,A0
trap #0
```

NAME

locking - provide exclusive file regions for reading or writing

SYNOPSIS

```
locking(fildes, mode, size)
int fildes;
int mode;
int size;
```

DESCRIPTION

Locking will allow a specified number of bytes to be accessed only by the locking process. Other processes which attempt to lock, read, or write the locked area will sleep until the area becomes unlocked.

Fildes is the word returned from a successful _open_, _creat_, _dup_, or _pipe_ system call.

Mode is zero to unlock the area. _Mode_ is one or two for making the area locked. If the _mode_ is one, and the area has some other lock on it, then the process will sleep until the entire area is available. If the _mode_ is two, and the area is locked, an error will be returned.

Size is the number of contigous bytes to be locked or unlocked. The area to be locked starts at the current offset in the file. If _size_ is zero the area to end of file is locked.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another processes locked area. Thus calls to _locking_, _read_, or _write_ scan for a deadlock prior to sleeping on a locked area. An error return is made if sleeping on the locked area would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked area for the same process. When this or adjacent areas occur, the areas are combined into a single area. If the request requires a new lock element with the lock table full, an error is returned, and the area is not locked.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional lock element to hold the cut off section. If the lock table is full, an error is returned, and the requested area is not released.

While locks may be applied to special files or pipes, read/write operations will not be blocked. Locks may not be applied to a directory.

SEE ALSO

open(2), creat(2), read(2), write(2), dup(2), close(2)

DIAGNOSTICS

The value -1 is returned if the file does not exist, or if a deadlock using file locks would occur. EACCES will be returned for lock requests in which the area is already locked by another process. EDEADLOCK will be returned by: read, write, or locking if a deadlock would occur. EDEADLOCK will also be returned when the locktable overflows.

ASSEMBLER

```
movl #45,D0
movl fildes,A0
movl mode,D1
movl size,A1
trap #0
```

Carry bit cleared on success.

NAME

lseek, tell - move read/write pointer

SYNOPSIS

```
long lseek(fildes, offset, whence)
int fildes;
long offset;
int whence;

long tell(fildes)
int fildes;
```

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 *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

The returned value is the resulting pointer location.

The function *tell*(*fildes*) is identical to *lseek*(*fildes*, *0L*, *1*).

Seeking far beyond the end of a file, then writing, creates a gap or 'hole', which occupies no physical space and reads as zeros.

SEE ALSO

open(2), creat(2), fseek(3)

DIAGNOSTICS

-1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file. The current file offset is returned.

BUGS

Lseek is a no-op on character special files.

ASSEMBLER

```
movl #19,D0
movl fildes,A0
movl offset,D1
movl whence,A1
trap #0
```

Carry bit cleared on success.

File offset returned in D0.

NAME

mknod - make a directory or a special file

SYNOPSIS

```
mknod(name, mode, addr)
char *name;
int mode;
int addr;
```

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by _name_. The mode of the new file (including directory and special file bits) is initialized from _mode_. (The protection part of the mode is modified by the process's mode mask; see _umask_(2)). The first block pointer of the i-node is initialized from _addr_. For ordinary files and directories _addr_ is normally zero. In the case of a special file, _addr_ specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(1), mknod(1), filsys(5)

DIAGNOSTICS

Zero is returned if the file has been made; -1 if the file already exists or if the user is not the super-user.

ASSEMBLER

```
movl #14,D0
movl #name,A0
movl mode,D1
movl addr,A1
trap #0
```

Carry bit cleared on success.

NAME

mount, umount - mount or remove a file system

SYNOPSIS

```
mount(special, name, rwflag)
char *special;
char *name;
int rwflag;

umount(special)
char *special;
```

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special*; from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. *Name* must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

SEE ALSO

mount(1), umount(1)

DIAGNOSTICS

Mount returns 0 if the action occurred; -1 if *special* is inaccessible or not an appropriate file; if *name* does not exist; if *special* is already mounted; if *name* is in use; or if there are already too many file systems mounted.

Umount returns 0 if the action occurred; -1 if if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

BUGS

If the file system is mounted on a directory having a mode that precludes user access (see *chmod*(*1*)), users may not be able to access the mounted file system. The directory will be able to be listed and everything will appear fine, including the access modes, but none of its files will be able to be accessed. Before mounting a file system on a directory, the super-user should check the protections on the directory to make sure that user access is permitted to the level desired.

If a file containing holes (unallocated blocks) is read, even on a file system mounted read-only, the system will attempt to fill in the holes by writing on the device.

ASSEMBLER

```
movl #21,D0        | sys mount
movl #special,A0
movl #name,D1
trap #0
```

Carry bit cleared on success.

```
movl #22,D0        | sys umount
movl #special,A0
trap #0
```

Carry bit cleared on success.

NAME

nice - set program priority

SYNOPSIS

```
nice(incr)
int incr;
```

DESCRIPTION

The scheduling priority of the process is augmented by _incr_. Positive priorities get less service than normal.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range 0 (most urgent) to 120 (least).

The priority of a process is passed to a child process by _fork_(2). For a privileged process to return to normal priority from an unknown state, _nice_ should be called with the argument -120 to change it from lowest priority to highest, no matter what priority it actually possessed. It should then be called with the argument 20 to get to the normal default priority.

EXAMPLE

```
nice(-120); nice(20);
```

would return you to the default priority.

SEE ALSO

fork(2), nice(1)

ASSEMBLER

```
movl #34,D0
movl incr,A0
trap #0
```

NAME

open - open for reading or writing

SYNOPSIS

```
open(name, mode)
char *name;
int mode;
```

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 file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

SEE ALSO

creat(2), read(2), write(2), dup(2), close(2)

DIAGNOSTICS

The value -1 is returned 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. writeable), or if too many files are open.

ASSEMBLER

```
movl #5,D0
movl #name,A0
movl mode,D1
trap #0
```

Carry bit cleared on success.

File descriptor is returned in D0.

NAME

pause - stop until signal

SYNOPSIS

```
pause()
```

DESCRIPTION

Pause never returns normally. It is used to give up control while waiting for a signal from *kill*(2) or *alarm*(2). Upon termination of a signal handler started during a *pause*, the *pause* call will return.

SEE ALSO

kill(1), kill(2), alarm(2), signal(2), setjmp(3)

ASSEMBLER

```
movl #29,D0
trap #0
```

NAME

phys - allow a process to access physical addresses

SYNOPSIS

```
phys(physnum, virtaddr, size, physaddr)
int physnum
char *virtaddr;
long size;
char *physaddr;
```

DESCRIPTION

The phys(2) call maps arbitrary physical memory into a process's virtual address space. _physnum_ is a number (0-3) that specifies which of 4 physical spaces to set up. Up to 4 phys(2) calls can be active at any one time. _virtaddr_ is the process's virtual address. _size_ is the number of bytes to map in. _physaddr_ is the physical address to map in.

Valid _virtaddr_ and _physaddr_ values are constrained by hardware and must be at an address multiple of the resolution of the CPU's memory management scheme. If _size_ is non zero, size is rounded up to the next MMU resolution boundary. If _size_ is zero, any previous phys(2) mapping for that _physnum_ segment is nullified.

For example, the call

phys(2, 0x100000, 32768, 0)

will allow a process to access physical locations 0 through 32767 by referencing virtual address 0x100000 through 0x100000+32767.

In actuality, the CPU MMU register is loaded with _physaddr_ shifted to account for page resolution.

phys(2) may only be executed by the super-user.

DIAGNOSTICS

The value zero is returned if the phys call was successful. The value -1 is returned if not super-user, if _virtaddr_ or _physaddr_ is not in the proper range, or if the specified _virtaddr_ segment register is already in use.

BUGS

This system call is very machine dependent.

ASSEMBLER

```
movl #52,D0
movl physnum,A0
movl #virtaddr,D1
movl size,A1
movl #physaddr,D2
trap #0
```

Carry bit cleared on success.

NAME

pipe - create an interprocess channel

SYNOPSIS

```
pipe(fildes)
int fildes[2];
```

DESCRIPTION

The _pipe_ system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor _fildes_[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor _fildes_[0] will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent _fork_ calls) will pass data through the pipe with _read_ and _write_ calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

SEE ALSO

sh(1), read(2), write(2), fork(2)

DIAGNOSTICS

The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

ASSEMBLER

```
movl #42,D0
movl #fildes,A0
trap #0
```

Carry bit cleared on success.

Read file descriptor in D0.
Write file descriptor in D1.

NAME

profil - execution time profile

SYNOPSIS

```
profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz;
int offset;
int 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, *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 factor is interpreted as an unsigned, short integer: in hex, FFFF(x) gives a 1-1 mapping of pc's to words in *buff*; 8000(x) maps each pair of instruction words together. 1(x) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling may be turned off if an update in *buff* would cause a memory fault.

SEE ALSO

monitor(3), prof(1)

ASSEMBLER

```
movl #44,D0
movl #buff,A0
movl bufsiz,D1
movl offset,A1
movl scale,D2
trap #0
```

NAME

ptrace - process trace

SYNOPSIS

```
#include <signal.h>

ptrace(request, pid, addr, data)
int request;
int pid;
int *addr;
int 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. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt.' See *signal*(2) for the list. Then the traced process enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned. *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. Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general

registers, the floating point status and registers, and certain bits of the processor status word.

7 The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the 'termination' status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

SEE ALSO

wait(2), signal(2), adb(1)

DIAGNOSTICS

The value -1 is returned if *request* is invalid, *pid* is not a traceable process, *addr* is out of bounds, or *data* specifies an illegal signal number.

BUGS

Ptrace is unique and arcane; it should be replaced with a special file which can be opened and read and written. The control functions could then be implemented with *ioctl*(2) calls on this file. This would be simpler to understand and have much higher performance.

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use 'illegal instruction' signals at a very high rate) could be efficiently debugged.

The error indication, -1, is a legitimate function value; *errno*, see *intro*(2), can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

ASSEMBLER

```
movl #26,D0
movl request,A0
movl pid,D1
movl #addr,A1
movl data,D2
trap #0
```

Carry bit cleared on success.

NAME

read - read from file

SYNOPSIS

```
read(fildes, buffer, nbytes)
int fildes;
char *buffer;
int nbytes;
```

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; if the file refers to a pipe, at least 1 byte and at most _nbytes_ will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open(2), creat(2), dup(2), pipe(2)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is -1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous _nbytes_, file descriptor not that of an input file.

ASSEMBLER

```
movl #3,D0
movl fildes,A0
movl #buffer,D1
movl nbytes,A1
trap #0
```

Carry bit cleared on success.

The number of bytes read is returned in D0.

NAME

setuid, setgid - set user and group ID

SYNOPSIS

```
setuid(uid)
int uid;

setgid(gid)
int gid;
```

DESCRIPTION

The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user or if the argument is the real or effective ID.

SEE ALSO

getuid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; -1 is returned otherwise.

ASSEMBLER

```
movl #23,D0        | sys setuid
movl uid,A0
trap #0
```

Carry bit cleared on success.

```
movl #46,D0        | sys setgid
movl gid,A0
trap #0
```

Carry bit cleared on success.

NAME

signal - catch or ignore signals

SYNOPSIS

```
#include &lt;signal.h&gt;

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

DESCRIPTION

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

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or link with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

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

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination, sometimes with a core image. If *func* is SIG_IGN the signal is ignored. Otherwise when the signal occurs *func* will be called with the signal number as argument. A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a *read* or *write*(2) on a slow device (like a typewriter; but not a file); and during *pause* or *wait*(2). When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of _signal_ is the previous (or initial) value of _func_ for the particular signal.

After a _fork_(2) the child inherits all signals. _Exec_(2) resets all caught signals to default action.

SEE ALSO

kill(1), kill(2), ptrace(2), setjmp(3)

DIAGNOSTICS

The value (int)-1 is returned if the given signal is out of range.

BUGS

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its _func_ argument are problematical.

ASSEMBLER

```
movl #48,D0
movl sig,A0
movl #func,D1
trap #0
```

Carry bit cleared on success.

The old value of the signal is returned in D0.

NAME

stat, fstat - get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

stat(name, buf)
char *name;
struct stat *buf;

fstat(fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

<u>Stat</u> obtains detailed information about a named file. <u>Fstat</u> obtains the same information about an open file known by the file descriptor from a successful <u>open</u>, <u>creat</u>, <u>dup</u> or <u>pipe</u>(2) call.

<u>Name</u> points to a null-terminated string naming a file; <u>buf</u> is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable.

The layout of the structure pointed to by buf as defined in <<u>stat.h</u>> is given below. "St_mode" is encoded according to the '#define' statements.

```
struct      stat
{
        dev_t             st_dev;
        ino_t             st_ino;
        unsigned short    st_mode;
        short             st_nlink;
        short             st_uid;
        short             st_gid;
        dev_t             st_rdev;
        off_t             st_size;
        time_t            st_atime;
        time_t            st_mtime;
        time_t            st_ctime;
};

#define     S_IFMT      0170000           /* type of file */
#define           S_IFDIR     0040000     /* directory */
#define           S_IFCHR     0020000     /* character special */
#define           S_IFBLK     0060000     /* block special */
#define           S_IFREG     0100000     /* regular */

#define     S_ISUID     0004000           /* set user id on execution */
#define     S_ISGID     0002000           /* set group id on execution */
```

```
#define   S_ISVTX   0001000   /* save swapped text even after use */
#define   S_IREAD   0000400   /* read permission, owner */
#define   S_IWRITE  0000200   /* write permission, owner */
#define   S_IEXEC   0000100   /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see chmod(2)). The defined types, ino_t, off_t, time_t, name various width integer values; dev_t encodes major and minor device numbers; their exact definitions are in the include file <sys/types.h> (see types(5).

When fildes is associated with a pipe, fstat reports an ordinary file with restricted permissions. The size is the number of bytes queued in the pipe.

st_atime is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. st_mtime is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. st_ctime is set both by writing and changing the i-node.

SEE ALSO

ls(1), filsys(5)

DIAGNOSTICS

Zero is returned if a status is available; -1 if the file cannot be found.

ASSEMBLER

```
movl #18,D0        | sys stat
movl #name,A0
movl #buf,D1
trap #0
```

Carry bit cleared on success.

```
movl #28,D0        | sys fstat
movl fildes,A0
movl #buf,D1
trap #0
```

Carry bit cleared on success.

NAME

stime - set time

SYNOPSIS

```
stime(tp)
long *tp;
```

DESCRIPTION

Stime sets the system's idea of the time and date. Time, pointed to by *tp*, is measured in seconds from 0000 GMT Jan 1, 1970. Only the super-user may use this call.

SEE ALSO

date(1), time(2), ctime(3)

DIAGNOSTICS

Zero is returned if the time was set; -1 if user is not the super-user.

ASSEMBLER

```
movl #25,D0
movl #tp,A0
trap #0
```

Carry bit cleared on success.

NAME

ioctl, stty, gtty - control device

SYNOPSIS

```
#include <sgtty.h>

ioctl(fildes, request, argp)
int fildes;
int request;
struct sgttyb *argp;

stty(fildes, argp)
int fildes;
struct sgttyb *argp;

gtty(fildes, argp)
int fildes;
struct sgttyb *argp;
```

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups on various devices in section 4 discuss how *ioctl* applies to them.

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *tty*(4).

The following two standard calls, however, apply to any open file:

```
ioctl(fildes, FIOCLEX, NULL);
ioctl(fildes, FIONCLEX, NULL);
```

The first causes the file to be closed automatically during a successful *exec* operation; the second reverses the effect of the first.

The following call also applies to any open file:

```
ioctl(fildes, FIONREAD, &count)
```

returning, in the longword *count* the number of characters available for reading from *fildes*.

SEE ALSO

stty(1), tty(4), exec(2)

DIAGNOSTICS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended, or if *request* attempts to modify the state of a terminal when *fildes* is not writeable.

BUGS

Strictly speaking, since *ioctl* may be extended in different ways to devices with different properties, *argp* should have an open-ended declaration like

```
union { struct sgttyb ...; ... } *argp;
```

The important thing is that the size is fixed by 'struct sgttyb'.

ASSEMBLER

See ioctl(2)

NAME

sync - update super-block

SYNOPSIS

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*, *fsck* etc. It is mandatory before bringing down the system.

SEE ALSO

sync(1M), update(1M)

BUGS

The writing, although scheduled, is not necessarily complete upon return from *sync*.

ASSEMBLER

```
movl 36,D0
trap #0
```

NAME

time, ftime - get date and time

SYNOPSIS

```
long time(0)

long time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>

ftime(tp)
struct timeb *tp;
```

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

The *ftime* entry fills in a structure pointed to by its argument, as defined by <*sys*/*timeb*.*h*>.

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

stime(2)

ASSEMBLER

```
movl #35,D0        | sys ftime
movl #tp,A0
trap #0

movl #13,D0        | sys time
movl #tloc,A0
trap #0
```

NAME

times - get process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>

times(buffer)
struct tms *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 (even in 50 Hz countries).

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

SEE ALSO

time(1), time(2),

ASSEMBLER

```
movl #43,D0
movl #buffer,A0
trap #0
```

NAME

umask - set file creation mode mask

SYNOPSIS

```
umask(complmode)
int complmode;
```

DESCRIPTION

Umask sets a mask used whenever a file is created by *creat*(2) or *mknod*(2): the actual mode (see *chmod*(2)) of the newly-created file is the difference between the given mode and *complmode*. Only the low-order 9 bits of *complmode* (the protection bits) participate. In other words, *complmode* shows the bits to be turned off when a new file is created.

The previous value of *complmode* is returned by the call. The value is initially 022, which is an octal "mask" number representing the complement of the desired mode. "022" here means that no permissions are withheld from the owner, but write permission is forbidden to group and to others. Its complement, the mode of the file, would be 755. *umask* is inherited by child processes.

SEE ALSO

creat(2), mknod(2), chmod(2)

ASSEMBLER

```
movl #60,D0
movl complmode,A0
trap #0
```

The previous value of umask is returned to D0.

NAME

unlink - remove directory entry

SYNOPSIS

```
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, any programs that already have the file open can continue to read or write it.

SEE ALSO

rm(1), link(2)

DIAGNOSTICS

Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

ASSEMBLER

```
movl #10,D0
movl #name,A0
trap #0
```

Carry bit cleared on success.

NAME

utime - set file times

SYNOPSIS

```
#include <sys/types.h>

utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

The *utime* call uses the "accessed" and "updated" times in that order from the *timep* vector to set the corresponding recorded times for the named *file*.

The caller must be the owner of the file or the super-user. The 'inode-changed' time of the file is set to the current time.

SEE ALSO

stat(2)

ASSEMBLER

```
movl #30,D0
movl #file,A0
movl #timep,D1
trap #0
```

NAME

wait - wait for process to terminate

SYNOPSIS

```
wait(status)
int *status;

wait(0)
```

DESCRIPTION

Wait causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last *wait*, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of -1 returned). The normal return yields the process ID of the terminated child. In the case of several children several *wait* calls are needed to learn of all the deaths.

If (int)*status* is nonzero, the next byte to the low byte of the word pointed to receives the low byte of the argument of *exit* when the child terminated. The low byte receives the termination status of the process. See *signal*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See *ptrace*(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit(2), fork(2), signal(2)

DIAGNOSTICS

Returns -1 if there are no children not previously waited for.

ASSEMBLER

```
movl #7,D0
movl #status,A0
trap #0
```

Process ID in D0.
Status in D1.

Carry flag is set if there are no children not previously waited for.

NAME

write - write on a file

SYNOPSIS

```
write(fildes, buffer, nbytes)
int fildes;
char *buffer;
int nbytes;
```

DESCRIPTION

A file descriptor is a word returned from a successful _open_, _creat_, _dup_, or _pipe_(2) call.

Buffer is the address of _nbytes_ contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat(2), open(2), pipe(2)

DIAGNOSTICS

Returns -1 on error: bad descriptor, buffer address, or count; physical I/O errors.

BUGS

No write errors to the file system are returned to the user.

ASSEMBLER

```
movl #4,D0
movl fildes,A0
movl #buffer,D1
movl nbytes,A1
trap #0
```

Carry bit cleared on success.

The number of bytes written is returned in D0.

NAME

intro - introduction to library functions

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions that may be found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in section 2. Functions are divided into various libraries distinguished by the section number at the top of the page:

(3) These functions, together with those of section 2 and those marked (3S), constitute library *libc*, which is automatically loaded by the C compiler *cc*(1). The link editor *ld*(1) searches this library under the '-lc' option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3M) These functions constitute the math library, *libm*. The link editor searches this library under the '-lm' option. Declarations for these functions may be obtained from the include file <math.h>.

(3S) These functions constitute the "standard I/O package", see *stdio*(3). These functions are in the library *libc* already mentioned. Declarations for these functions may be obtained from the include file <stdio.h>.

(3X) Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

FILES

/lib/libc.a
/lib/libm.a, /usr/lib/libm.a (one or the other)

SEE ALSO

stdio(3), cc(1), intro(2), ld(1), nm(1)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE. The values of EDOM and ERANGE are defined in the include file <*math.h*>.

NAME

abort - generate a fault

SYNOPSIS

abort()

DESCRIPTION

Abort executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may subsequently be used for debugging.

SEE ALSO

adb(1), signal(2), exit(2)

DIAGNOSTICS

Usually "TRACE/BPT trap - Core dumped" from the shell.

NAME
abs - integer absolute value

SYNOPSIS
```
abs(i)
int i;
```

DESCRIPTION
Abs returns the absolute value of its integer operand.

SEE ALSO
floor(3) for *fabs*

BUGS
You get what the hardware gives on the smallest integer.

NAME

atof, atoi, atol - convert ASCII to numbers

SYNOPSIS

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION

These functions convert a string pointed to by _nptr_ to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

Atoi and _atol_ recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3)

BUGS

There are no provisions for overflow.

NAME

crypt, setkey, encrypt - DES encryption

SYNOPSIS

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
int edflag;
```

DESCRIPTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The other entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the *encrypt* entry is likewise a character array of length 64 containing 0's and 1's. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is 0, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

passwd(1), passwd(5), login(1), getpass(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ctime, localtime, gmtime, asctime, timezone - convert date and time to ASCII

SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
int zone, dst;
```

DESCRIPTION

Ctime converts a time pointed to by *clock* such as returned by *time*(2) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

Localtime and *gmtime* return pointers to structures containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. *Asctime* converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm { /* see ctime(3) */
        int   tm_sec;
        int   tm_min;
        int   tm_hour;
        int   tm_mday;
        int   tm_mon;
        int   tm_year;
        int   tm_wday;
        int   tm_yday;
        int   tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the standard U.S.A. daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

Timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g. in Afghanistan *timezone*(-(*60***4*+*30*), *0*) is appropriate because it is 4:30 ahead of GMT and the string GMT+4:30 is produced.

SEE ALSO

time(2)

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii - character classification

SYNOPSIS

```
#include <ctype.h>

isalpha(c)

. . .
```

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. _Isascii_ is defined on all integer values; the rest are defined only where _isascii_ is true and on the single non-ASCII value EOF (see _stdio_(3)).

isalpha	_c_ is a letter
isupper	_c_ is an upper case letter
islower	_c_ is a lower case letter
isdigit	_c_ is a digit
isalnum	_c_ is an alphanumeric character
isspace	_c_ is a space, tab, carriage return, newline, or formfeed
ispunct	_c_ is a punctuation character (neither control nor alphanumeric)
isprint	_c_ is a printing character, code 040(8) (space) through 0176 (tilde)
iscntrl	_c_ is a delete character (0177) or ordinary control character (less than 040).
isascii	_c_ is an ASCII character, code less than 0200

SEE ALSO

ascii(7)

NAME

curses - screen functions with "optimal" cursor motion

SYNOPSIS

cc [flags] files -lcurses -ltermcap [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh*() tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr*() must be called before any of the other routines that deal with windows and screens are used. The routine *endwin*() should be called before exiting.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold,
stty(2), setenv(3), termcap(5)

AUTHOR

Ken Arnold (U.C. Berkeley)

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(win)	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(name)	get terminal capability *name*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinates
initscr()	initialize screens
insch(c)	insert a char
insertln()	insert a line
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window

nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on stdscr
raw()	set raw mode
refresh()	make current screen look like stdscr
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through stdscr
scroll(win)	scroll win one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(win,lines,cols,begin_y,begin_x)	create a subwindow
touchwin(win)	change all of win
unctrl(ch)	printable version of ch
waddch(win,ch)	add char to win
waddstr(win,str)	add string to win
wclear(win)	clear win
wclrtobot(win)	clear to bottom of win
wclrtoeol(win)	clear to end of line on win
wdelch(win,c)	delete char from win
wdeleteln(win)	delete line from win
werase(win)	erase win
wgetch(win)	get a char through win
wgetstr(win,str)	get a string through win
winch(win)	get char at current (y,x) in win
winsch(win,c)	insert char into win
winsertln(win)	insert line into win
wmove(win,y,x)	set current (y,x) co-ordinates on win
wprintw(win,fmt,arg1,arg2,...)	printf on win
wrefresh(win)	make screen look like win
wscanw(win,fmt,arg1,arg2,...)	scanf through win
wstandend(win)	end standout mode on win
wstandout(win)	start standout mode on win

NAME

ecvt, fcvt, gcvt - output conversion

SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

DESCRIPTION

Ecvt converts the _value_ to a null-terminated string of _ndigit_ ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through _decpt_ (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by _sign_ is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to _ecvt_, except that the correct digit has been rounded.

Gcvt converts the _value_ to a null-terminated ASCII string in _buf_ and returns a pointer to _buf_. It attempts to produce _ndigit_ significant digits in E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3)

BUGS

The return values point to static data whose content is overwritten by each call.

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. The address of "etext" is the first address above the program text, "edata" above the initialized data region, and "end" above the uninitialized data region.

When execution begins, the program break coincides with "end", but it is reset by the routines _brk_(2), _malloc_(3), standard input/output (_stdio_(3)), the profile (-p) option of _cc_(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see _brk_(2).

SEE ALSO

brk(2), malloc(3)

NAME

exp, log, log10, pow, sqrt - exponential, logarithm, power, square root

SYNOPSIS

```
#include &lt;math.h&gt;

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

DESCRIPTION

Exp returns the exponential function of *x*.

Log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

Pow returns x^y, x to the y power.

Sqrt returns the square root of *x*.

SEE ALSO

hypot(3), sinh(3), intro(2)

DIAGNOSTICS

Exp and *pow* return a huge value when the correct value would overflow; *errno* is set to ERANGE. *Pow* returns 0 and sets *errno* to EDOM when the second argument is negative and non-integral and when both arguments are 0.

Log returns 0 when *x* is zero or negative; *errno* is set to EDOM.

Sqrt returns 0 when *x* is negative; *errno* is set to EDOM.

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

Fclose causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed to be used with another *fopen*.

Fclose is performed automatically upon calling *exit*(2).

Fflush causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

SEE ALSO

close(2), fopen(3), setbuf(3)

DIAGNOSTICS

These routines return EOF if *stream* is not associated with an output file, or if buffered data cannot be transferred to that file.

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;
```

DESCRIPTION

Feof returns non-zero when end of file is read on the named input _stream_, otherwise zero.

Ferror returns non-zero when an error has occurred reading or writing the named _stream_, otherwise zero. Unless cleared by _clearerr_, the error indication lasts until the stream is closed.

Clrerr resets the error indication on the named _stream_.

Fileno returns the integer file descriptor associated with the _stream_, see _open_(2).

These functions are presently implemented as macros in <stdio.h>; they cannot be redeclared.

SEE ALSO

fopen(3), open(2)

NAME

fabs, floor, ceil - absolute value, floor, ceiling functions

SYNOPSIS

```
#include &lt;math.h&gt;

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

Fabs returns the absolute value |_x_|.

Floor returns the largest integer not greater than _x_.

Ceil returns the smallest integer not less than _x_.

SEE ALSO

abs(3)

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by _filename_ and associates a stream with it. _Fopen_ returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

"r" open for reading

"w" create for writing

"a" append: open for writing at end of file, or create for writing

In addition, each _type_ may be followed by a '+' to have the file opened for reading and writing. "r+" positions the stream at the beginning of the file, "w+" creates or truncates it, and "a+" positions it at the end. Both reads and writes may be used on read/write streams, with the limitation that an _fseek_, _rewind_, or reading an end-of-file must be used between a read and a write or vice-versa.

Freopen substitutes the named file in place of the open _stream_. It returns the original value of _stream_. The original stream is closed.

Freopen is typically used to attach the preopened constant names, stdin, stdout, stderr, to specified files.

Fdopen associates a stream with a file descriptor obtained from _open_, _dup_, _creat_, or _pipe_(2). The _type_ of the stream must agree with the mode of the open file.

SEE ALSO

open(2), fclose(3)

DIAGNOSTICS

Fopen and _freopen_ return the pointer NULL if _filename_ cannot be accessed.

BUGS

Fdopen is not portable to systems other than UNIX.

The read/write *types* do not exist on all systems. Those systems without read/write modes will probably treat the *type* as if the '+' was not present.

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
int ptr;
int nitems; FILE *stream;

fwrite(ptr, sizeof(*ptr), nitems, stream)
int ptr;
int nitems; FILE *stream;
```

DESCRIPTION

Fread reads, into a block beginning at _ptr_, _nitems_ of data of the type of *_ptr_ from the named input _stream_. It returns the number of items actually read.

If _stream_ is stdin and the standard output is line buffered, then any partial output line will be flushed before any call to _read_(2) to satisfy the _fread_.

Fwrite appends at most _nitems_ of data of the type of *_ptr_ beginning at _ptr_ to the named output _stream_. It returns the number of items actually written.

SEE ALSO

read(2), write(2), fopen(3), getc(3), putc(3), gets(3), puts(3), printf(3), scanf(3)

DIAGNOSTICS

Fread and _fwrite_ return 0 upon end of file or error.

NAME

frexp, ldexp, modf - split into mantissa and exponent

SYNOPSIS

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

Frexp returns the mantissa of a double *value* as a double quantity, *x*, of magnitude less than 1 and stores an integer *n* such that *value* = *x*$*2^{n}$ indirectly through *eptr*.

Ldexp returns the quantity *value*$*2^{exp}$.

Modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

NAME

fseek, ftell, rewind - reposition a stream

SYNOPSIS

```
#include &lt;stdio.h&gt;

fseek(stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell(stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according to whether *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc*(3).

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on UNIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an *offset* for *fseek*.

Rewind(*stream*) is equivalent to *fseek*(*stream*, *0L*, *0*).

SEE ALSO

lseek(2), fopen(3)

DIAGNOSTICS

Fseek returns -1 for improper seeks.

NAME

getc, getchar, fgetc, getw - get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character from the named input *stream*.

Getchar() is identical to *getc*(*stdin*).

Fgetc behaves like *getc*, but is a genuine function, not a macro; it may be used to save object text.

Getw returns the next word (32-bit integer on a 68000) from the named input *stream*. It returns the constant EOF upon end of file or error, but since that is a good integer value, *feof* and *ferror*(3) should be used to check the success of *getw*. *Getw* assumes no special alignment in the file.

SEE ALSO

fopen(3), putc(3), gets(3), scanf(3), fread(3), ungetc(3)

DIAGNOSTICS

These functions return the integer constant EOF at end of file or upon read error.

A stop with message, "Reading bad file", means an attempt has been made to read from a stream that has not been opened for reading by *fopen*.

BUGS

The end-of-file return from *getchar* is incompatible with that in UNIX editions 1-6.

Because it is implemented as a macro, *getc* treats a *stream* argument with side effects incorrectly. In particular, "getc(*f++);" doesn't work sensibly.

NAME

getenv - value for environment name

SYNOPSIS

```
char *getenv(name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value* and returns pointer to *value* string in the environment if such a string is present, otherwise 0 (NULL).

SEE ALSO

environ(5), exec(2)

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent - get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent()

struct group *getgrgid(gid)
int gid;

struct group *getgrnam(name)
char *name;

setgrent()

endgrent()
```

DESCRIPTION

Getgrent, _getgrgid_ and _getgrnam_ each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

```
        struct      group {
                char  *gr_name;
                char  *gr_passwd;
                int   gr_gid;
                char  **gr_mem;
        };
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group-ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

Getgrent simply reads the next line while _getgrgid_ and _getgrnam_ search until a matching _gid_ or _name_ is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

A call to _setgrent_ has the effect of rewinding the group file to allow repeated searches. _Endgrent_ may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3), getpwent(3), group(5)

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getlogin - get login name

SYNOPSIS

```
char *getlogin()
```

DESCRIPTION

Getlogin returns a pointer to the login name as found in /_etc_/_utmp_. It may be used in conjunction with _getpwnam_ to locate the correct password file entry when the same userid is shared by several login names.

If _getlogin_ is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to first call _getlogin_ and if it fails, to call _getpwuid_.

FILES

/etc/utmp

SEE ALSO

getpwent(3), getgrent(3), utmp(5)

DIAGNOSTICS

Null pointer (0) returned if name could not be found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getpass - read a password

SYNOPSIS

```
char *getpass(prompt)
char *prompt;
```

DESCRIPTION

Getpass reads a password from the file */dev/tty*, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

/dev/tty

SEE ALSO

crypt(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw - get name from uid

SYNOPSIS

```
getpw(uid, buf)
int uid;
char *buf;
```

DESCRIPTION

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent - get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

int setpwent()

int endpwent()
```

DESCRIPTION

Getpwent, *getpwuid* and *getpwnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

```
        struct      passwd { /* see getpwent(3) */
                char    *pw_name;
                char    *pw_passwd;
                int     pw_uid;
                int     pw_gid;
                int     pw_quota;
                char    *pw_comment;
                char    *pw_gecos;
                char    *pw_dir;
                char    *pw_shell;
        };
```

The fields *pw quota* and *pw comment* are unused; the others have meanings described in *passwd*(5).

Getpwent reads the next line (opening the file if necessary); *setpwent* rewinds the file; *endpwent* closes it.

Getpwuid and *getpwnam* search from the beginning until a matching *uid* or *name* is found (or until EOF is encountered).

FILES

/etc/passwd

SEE ALSO

getlogin(3), getgrent(3), passwd(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Gets reads a string into *s* from the standard input stream stdin. The string is terminated by a newline character, which is replaced in *s* by a null character. *Gets* returns its argument.

Fgets reads *n*-1 characters, or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. *Fgets* returns its first argument.

SEE ALSO

puts(3), getc(3), scanf(3), fread(3), ferror(3)

DIAGNOSTICS

Gets and *fgets* return the constant pointer NULL upon end of file or error.

BUGS

Gets deletes a newline, *fgets* keeps it, all in the name of backward compatibility.

NAME

hypot, cabs - Euclidean distance

SYNOPSIS

```
#include &lt;math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;
```

DESCRIPTION

Hypot and *cabs* return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

exp(3) for *sqrt*

NAME

isatty - find name of a terminal

SYNOPSIS

```
isatty(fildes)
int fildes;
```

DESCRIPTION

Isatty returns 1 if _fildes_ is associated with a terminal device, 0 otherwise.

FILES

/dev/*

SEE ALSO

ioctl(2), ttys(5)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

j0, j1, jn, y0, y1, yn - bessel functions

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
int n;
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
int n;
double x;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause *y0*, *y1*, and *yn* to return a huge negative value and set *errno* to EDOM.

NAME

malloc, free, realloc, calloc - main memory allocator

SYNOPSIS

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and _free_ provide a simple general-purpose memory allocation package. _Malloc_ returns a pointer to a block of at least _size_ bytes beginning on a word boundary.

The argument to _free_ is a pointer to a block previously allocated by _malloc_; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by _malloc_ is overrun or if some random number is handed to _free_.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls _sbrk_ (see _break_(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by _ptr_ to _size_ bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Realloc also works if _ptr_ points to a block freed since the last call of _malloc_, _realloc_ or _calloc_; thus sequences of _free_, _malloc_ and _realloc_ can exploit the search strategy of _malloc_ to do storage compaction.

Calloc allocates space for an array of _nelem_ elements of size _elsize_. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, _realloc_ and _calloc_ return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. _Malloc_ may be recompiled to check the arena very stringently on every transaction; see the source code.

BUGS

When _realloc_ returns 0, the block pointed to by _ptr_ may be destroyed.

The current incarnation of the allocator is unsuitable for direct use in a large virtual environment where many small blocks are to be kept, since it keeps all allocated and freed blocks on a single circular list. Just before more memory is allocated, all allocated and freed blocks are referenced; this can cause a huge number of page faults.

NAME

mktemp - make a unique file name

SYNOPSIS

```
char *mktemp(template)
char *template;
```

DESCRIPTION

Mktemp replaces _template_ by a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with the current process id and a unique letter.

SEE ALSO

getpid(2)

NAME

monitor - prepare execution profile

SYNOPSIS

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

DESCRIPTION

An executable program created by "cc -p" automatically includes calls for _monitor_ with default parameters; _monitor_ needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to _profil_(2). _Lowpc_ and _highpc_ are the addresses of two functions; _buffer_ is the address of a (user supplied) array of _bufsize_ short integers. _Monitor_ arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of _lowpc_ and the highest is just below _highpc_. At most _nfunc_ call counts can be kept; only calls of functions compiled with the profiling option -p of _cc_(1) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext();
. . .
monitor((int) 2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text, see _end_(3).

To stop execution monitoring and write the results on the file _mon.out_, use

```
monitor(0);
```

then _prof_(1) can be used to examine the results.

FILES

mon.out

SEE ALSO

prof(1), profil(2), cc(1)

NAME

nlist - get entries from name list

SYNOPSIS

```
#include <a.out.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];
```

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(5) for the structure declaration.

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(5)

DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

perror, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

DESCRIPTION

Perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno* (see *intro*(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings "sys_errlist" is provided; *errno* can be used as an index in this table to get the message string without the newline. "Sys_nerr" is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2)

NAME

popen, pclose - initiate I/O to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;
```

DESCRIPTION

The arguments to _popen_ are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by _popen_ should be closed by _pclose_, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

SEE ALSO

pipe(2), fopen(3), fclose(3), system(3), wait(2)

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or the Shell cannot be accessed.

Pclose returns -1 if _stream_ is not associated with a 'popened' command.

BUGS

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with _fflush_, see _fclose_(3).

NAME

printf, fprintf, sprintf - formatted output conversion

SYNOPSIS

```
#include &lt;stdio.h&gt;

printf(format [, arg ] ... )
char *format;

fprintf(stream, format [, arg ] ... )
FILE *stream;
char *format;

sprintf(s, format [, arg ] ... )
char *s, *format;
```

DESCRIPTION

Printf places output on the standard output stream _stdout_. _Fprintf_ places output on the named output _stream_. _Sprintf_ places 'output' in the string _s_, followed by the character '\0'.

Each of these functions converts, formats, and prints its arguments after the 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 _arg printf_.

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign '-' which specifies _left adjustment_ of the converted value in the indicated field;

- an optional digit string specifying a _field width_; if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;

- an optional period '.' which serves to separate the field width from the next digit string;

- an optional digit string specifying a _precision_ which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;

- the character l specifying that a following d, o, x, or u corresponds to a long integer _arg_. (A capitalized conversion code accomplishes the same thing.)

- a character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer _arg_ supplies the field width or precision.

The conversion characters and their meanings are

dox The integer _arg_ is converted to decimal, octal, or hexadecimal notation respectively.

f The float or double _arg_ is converted to decimal notation in the style "[-]ddd.ddd" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e The float or double _arg_ is converted in the style "[-]d.ddde±dd" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double _arg_ is printed in style d, in style f, or in style e, whichever gives full precision in minimum space.

c The character _arg_ is printed. Null characters are ignored.

s _Arg_ is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

u The unsigned integer _arg_ is converted to decimal and printed (the result will be in the range 0 to 2**32-1

% Print a '%'; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by _printf_ are printed by _putc_(3).

Examples
To print a date and time in the form 'Sunday, July 3, 10:02', where _weekday_ and _month_ are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

To print pi to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

putc(3), scanf(3), ecvt(3)

BUGS

Very wide fields (>128 characters) fail.

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)
char c;

fputc(c, stream)
char c;
FILE *stream;

putw(w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc appends the character _c_ to the named output _stream_. It returns the character written.

Putchar(_c_) is defined as _putc_(_c_, _stdout_).

Fputc behaves like _putc_, but is a genuine function rather than a macro. It may be used to save on object text.

Putw appends word (i.e. _int_ of 32 bits on the 68000) "w" to the output _stream_. It returns the word written. _Putw_ neither assumes nor causes special alignment in the file.

The standard stream _stdout_ is normally buffered if and only if the output does not refer to a terminal; this default may be changed by _setbuf_(3). The standard stream _stderr_ is by default unbuffered unconditionally, but use of _freopen_ (see _fopen_(3)) will cause it to become buffered; _setbuf_, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. _Fflush_ (see _fclose_(3)) may be used to force the block out early.

SEE ALSO

fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

DIAGNOSTICS

These functions return the constant EOF upon error. Since this is a good integer, _ferror_(3) should be used to detect _putw_ errors.

BUGS

Because it is implemented as a macro, *putc* treats a *stream* argument with side effects improperly. In particular "putc(c, *f++);" doesn't work sensibly.

Errors can occur long after the call to *putc*.

NAME

puts, fputs - put a string on a stream

SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Puts copies the null-terminated string _s_ to the standard output stream _stdout_ and appends a newline character.

Fputs copies the null-terminated string _s_ to the named output _stream_.

Neither routine copies the terminal null character.

SEE ALSO

fopen(3), gets(3), putc(3), printf(3), ferror(3)
fread(3)

BUGS

Puts appends a newline, _fputs_ does not, all in the name of backward compatibility.

NAME

qsort - quicker sort

SYNOPSIS

```
qsort(base, nel, width, compar)
char *base;
int nel;
int width;
int (*compar)();
```

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared.

The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

```
struct entry {
      char   *name;
      int    flags;
};

main()
{
      struct entry hp[100];

      for (i = 0; i < count; i++ {
            /* fill the structure with the name and flags */
            .
            .
            .
      }
      qsort(hp, count, sizeof hp[0], entcmp);
}

entcmp(ep,ep2)
struct entry *ep, *ep2;
{
      return (strcmp(ep->name, ep2->name));
}
```

will sort a set of names with associated flags in ascii order. This example was taken from diffdir.

SEE ALSO

sort(1)

NAME

rand, srand - random number generator

SYNOPSIS

```
srand(seed)
int seed;

rand()
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator with period 2**32 to return successive pseudo-random numbers in the range from 0 to 2**31-1.

The generator is reinitialized by calling *srand* with 1 as argument. It can be set to a random starting point by calling *srand* with whatever you like as argument.

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include <stdio.h>

scanf(format [ , pointer ] . . . )
char *format;

fscanf(stream, format [ , pointer ] . . . )
FILE *stream;
char *format;

sscanf(s, format [ , pointer ] . . . )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream _stdin_. _Fscanf_ reads from the named input _stream_. _Sscanf_ reads from the character string _s_. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string _format_, described below, and a set of _pointer_ arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.

2. An ordinary character (not %) which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

% a single '%' is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

o an octal integer is expected; the corresponding argument should be a integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `\0', which will be added. The input field is terminated by a space character or a newline.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try "%1s". If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

f a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a _float_. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o and x may be capitalized or preceeded by l to indicate that a pointer to long rather than to int is in the argument list. Similarly, the conversion characters e or f may be capitalized or preceded by l to indicate a pointer to double rather than to float. The conversion characters d, o and x may be preceeded by h to indicate a pointer to short rather than to int.

The _scanf_ functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant EOF is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E-1  thompson
```

will assign to _i_ the value 25, _x_ the value 5.432, and _name_ will contain '_thompson\0_'. Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to _i_, 789.0 to _x_, skip '0123', and place the string '56\0' in _name_. The next call to _getchar_ will return 'a'.

SEE ALSO

atof(3), getc(3), printf(3)

DIAGNOSTICS

The _scanf_ functions return EOF on end of input, and a short count for missing or illegal data items.

BUGS

The success of literal matches and suppressed assignments is not directly determinable.

NAME

setbuf - assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from *malloc*(3) upon the first *getc* or *putc*(3) on the file, except that the standard output is line buffered when directed to a terminal. Other output streams directed to terminals, and the standard error stream *stderr* are normally not buffered. If the standard output is line buffered, then it is flushed each time data is read from the standard input by *read*(2).

SEE ALSO

fopen(3), getc(3), putc(3), malloc(3)

BUGS

The standard error stream should be line buffered by default.

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
int val;
```

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in _env_ for later use by _longjmp_. It returns value 0.

Longjmp restores the environment saved by the last call of _setjmp_. It then returns in such a way that execution continues as if the call of _setjmp_ had just returned the value _val_ to the function that invoked _setjmp_, which must not itself have returned in the interim. All accessible register variables and local data have values as of the time _longjmp_ was called.

SEE ALSO

signal(2)

NAME

sin, cos, tan, asin, acos, atan, atan2 - trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

DESCRIPTION

Sin, _cos_ and _tan_ return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sin in the range -pi/2 to pi/2.

Acos returns the arc cosine in the range 0 to pi.

Atan returns the arc tangent of _x_ in the range -pi/2 to pi/2.

Atan2 returns the arc tangent of _x_/_y_ in the range -pi to pi.

DIAGNOSTICS

Arguments of magnitude greater than 1 cause _asin_ and _acos_ to return value 0; _errno_ is set to EDOM. The value of _tan_ at its singular points is a huge number, and _errno_ is set to ERANGE.

BUGS

The value of _tan_ for arguments greater than about 2**31 is garbage.

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

```
#include &lt;math.h&gt;

double sinh(x)

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

DIAGNOSTICS

Sinh and _cosh_ return a huge value of appropriate sign when the correct value would overflow.

NAME

sleep - suspend execution for interval

SYNOPSIS

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an alarm clock signal and pausing until it occurs. The previous state of this signal is saved and restored. If the sleep time exceeds the time to the alarm signal, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

SEE ALSO

alarm(2), pause(2)

NAME

stdio - standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;
```

DESCRIPTION

These functions constitute an efficient user-level buffering scheme. The in-line macros _getc_ and _putc_(3) handle characters quickly. The higher level routines _gets_, _fgets_, _scanf_, _fscanf_, _fread_, _puts_, _fputs_, _printf_, _fprintf_, _fwrite_ all use _getc_ and _putc_; they can be freely intermixed.

A file with associated buffering is called a _stream_, and is declared to be a pointer to a defined type FILE. _Fopen_(3) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file

A constant 'pointer' NULL (0) designates no stream at all.

An integer constant EOF (-1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file <stdio.h> of pertinent macro definitions. The functions and constants mentioned in Section 3 are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: _getc_, _getchar_, _putc_, _putchar_, _feof_, _ferror_, _fileno_.

SEE ALSO

open(2), close(2), read(2), write(2)

DIAGNOSTICS

The value EOF is returned uniformly to indicate that a FILE pointer has not been initialized with _fopen_, input (output) has been attempted on an output (input) stream, or a FILE pointer designates corrupt or otherwise unintelligible FILE data.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to _fflush_(3) the standard output before going off and computing so that the output will appear.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - string operations

SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;
int n;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;
int n;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;
int n;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string _s2_ to the end of string _s1_. _Strncat_ copies at most _n_ characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as _s1_ is lexicographically greater than, equal to, or less than _s2_. _Strcmp_ uses native character comparison, which is signed. _Strncmp_ makes the same comparison but looks at at most _n_ characters.

Strcpy copies string _s2_ to _s1_, stopping after the null character has been moved. _Strncpy_ copies exactly _n_ characters, truncating or null-padding _s2_; the target may not be null-terminated if the length of _s2_ is _n_ or more. Both return _s1_.

Strlen returns the number of non-null characters in _s_.

Index (_rindex_) returns a pointer to the first (last) occurrence of character _c_ in string _s_, or zero if _c_ does not occur in the string.

NAME

swab - swap bytes

SYNOPSIS

```
swab(from, to, nbytes)
char *from, *to;
int nbytes;
```

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between to other machines. *Nbytes* should be even.

NAME

system - issue a shell command

SYNOPSIS

```
system(string)
char *string;
```

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

popen(3), exec(2), wait(2)

DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

BUGS

There should be a way to specify a shell other than *sh*(*1*).

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;
int destcol;
int destline;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base _termcap_(5). These are low level routines; see _curses_(3) for a higher level package.

Tgetent extracts the entry for terminal _name_ into the buffer at _bp_. _Bp_ should be a character buffer of size 1024 and must be retained through all subsequent calls to _tgetnum_, _tgetflag_, and _tgetstr_. _Tgetent_ returns -1 if it cannot open the _termcap_ file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than /_etc_/_termcap_. This can speed up entry into programs that call _tgetent_, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file /_etc_/_termcap_.

Tgetnum gets the numeric value of capability _id_, returning -1 if is not given for the terminal. _Tgetflag_ returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. _Tgetstr_ gets the string value of capability _id_, placing it in the buffer at _area_, advancing the _area_ pointer. It decodes the abbreviations for this field described in _termcap_(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from _cm_ to go to column _destcol_ in line _destline_. It uses the external variables UP (from the up capability) and BC (if bc is given rather than bs) if necessary to avoid placing \n, ^D or ^@ in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since tgoto may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then _tgoto_ returns OOPS.

Tputs decodes the leading padding information of the string _cp_; _affcnt_ gives the number of lines affected by the operation, or 1 if this is not applicable, _outc_ is a routine which is called with each character in turn. The external variable _ospeed_ should contain the output speed of the terminal as encoded by _stty_ (_2_). The external variable PC should contain a pad character to be used (from the pc capability) if a null (^@) is inappropriate.

FILES

usr/lib/libtermcap.a	termcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

AUTHOR

William Joy

NAME

ttyname, ttyslot - find name of a terminal

SYNOPSIS

```
char *ttyname(fildes)
int fildes;

ttyslot()
```

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Ttyslot returns the number of the entry in the *ttys*(5) file for the control terminal of the current process.

FILES

/dev/*
/etc/ttys

SEE ALSO

ioctl(2), isatty(3), ttys(5)

DIAGNOSTICS

Ttyname returns a null pointer (0) if *fildes* does not describe a terminal device in directory '/dev'.

Ttyslot returns 0 if '/etc/ttys' is inaccessible or if it cannot determine the control terminal.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ungetc - push character back into input stream

SYNOPSIS

```
#include <stdio.h>

ungetc(c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *Ungetc* returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

Fseek(3) erases all memory of pushed back characters.

SEE ALSO

getc(3), setbuf(3), fseek(3)

DIAGNOSTICS

Ungetc returns EOF if it can't push a character back.

NAME

intro - introduction to special files

DESCRIPTION

This section describes the special files and related driver functions available on the system.

NAME

mem, kmem - main memory

DESCRIPTION

Mem is a special file that is an image of the main memory of the computer. It may be used, for example, to examine (and even to patch) the system.

Byte addresses in *mem* are interpreted as physical memory addresses. 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.

FILES

/dev/mem, /dev/kmem

BUGS

Memory files are accessed in an inappropriate method for some device registers.

NAME
null - data sink

DESCRIPTION
Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES
/dev/null

NAME

tty - general terminal interface

DESCRIPTION

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

When a terminal file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by _init_(_1M_) and become a user's standard input and standard output device. The very first terminal file open in a process becomes the _control terminal_ for that process. The control terminal plays a special role in handling quit or interrupt signals, as discussed below. The control terminal is inherited by a child process during a _fork_, even if the control terminal is closed. The set of processes that thus share a control terminal is called a _process group_; all members of a process group receive certain signals together, see DEL below and _kill_(2).

The file /_dev_/_tty_ is, in each process, a synonym for the control terminal associated with that process. The above-mentioned /dev/tty file is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use. [The terminals associated with various processes can, if needed, be discovered using _ps_(_1_)].

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information. There are special modes, discussed below, that permit the program to read each character as typed without waiting for a full line. Certain ASCII control characters have special meaning. These characters are not passed to a reading program except in "raw" mode where they lose their special character. Also, it is possible to change these characters from the default; see below.

EOT (Control-D) may be used to generate an end of file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication.

DEL (Rubout) is not passed to a program but generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See *signal*(2).

FS (Control-\ or control-shift-L) generates the *quit* signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated.

DC3 (Control-S) delays all printing on the terminal until something is typed in.

DC1 (Control-Q) restarts printing after DC3 without generating any input to a program.

During input, erase and kill processing is normally done. By default, the character "^H" (control-h) erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. By default, the character '@' 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 "^H" may be entered literally by preceding it by a backslash '\'; the erase or kill character remains, but the '\' disappears. These two characters may be changed to others.

On input, when desired, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. In addition, the following escape sequences can be generated on output and accepted on input:

for	use
`	\'
|	\!
~	\^
{	\(
}	\)

When one or more characters are sent by the system to a user, 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 usually generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals that respond to it from hanging up.

Several _ioctl_(2) calls apply to terminals. Most of them use the following structure, defined in <_sgtty.h_>:

```
struct sgttyb {
      char  sg_ispeed;
      char  sg_ospeed;
      char  sg_erase;
      char  sg_kill;
      int   sg_flags;
};
```

The "sg_ispeed" and "sg_ospeed" fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <_sgtty.h_>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
B19200	14	19200 baud
EXTA	14	External A
EXTB	15	External B

The "sg_erase" and "sg_kill" fields of the argument structure specify the erase and kill characters respectively. (Defaults are Control H (backspace)and @.)

The "sg_flags" field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	

CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

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

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

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

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

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

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file indicator (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially. There are no delays and no echoing, and no replacement of one character for another; characters are a full 8 bits for both input and output (parity is up to the program).

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed.

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line, but quit and interrupt work, and output delays, case-translation, CRMOD, XTABS, ECHO, and parity work normally. On the other hand there is no erase or kill, and no special treatment of \ or EOT.

TANDEM mode causes the system to produce a stop character (default DC3) whenever the input queue is in danger of overflowing, and a start character (default DC1) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is actually another machine that obeys the conventions.

Several _ioctl_ calls have the form:

```
#include <sgtty.h>

ioctl(fildes, code, arg)
int fildes, code;
struct sgttyb *arg;
```

The applicable codes are:

TIOCGETP
: Fetch the parameters associated with the terminal, and store in the pointed-to structure.

TIOCSETP
: Set the parameters according to the pointed-to structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN
: Set the parameters but do not delay or flush input. Switching out of RAW or CBREAK mode may cause some garbage input.

With the following codes the _arg_ is ignored.

TIOCEXCL
: Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL
: Turn off "exclusive-use" mode.

TIOCHPCL
: When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH
: All characters waiting in input or output queues are flushed.

FIONREAD
: Return the number of characters in a terminal's input buffer into the integer pointer *arg.

The following codes affect characters that are special to the terminal interface. The argument is a pointer to the following structure, defined in <sgtty.h>:

```
struct tchars {
      char  t_intrc;          /* interrupt */
      char  t_quitc;          /* quit */
      char  t_startc;         /* start output */
      char  t_stopc;          /* stop output */
      char  t_eofc;           /* end-of-file */
      char  t_brkc;           /* input delimiter (like nl) */
};
```

The default values for these characters are DEL, FS, DC1, DC3, EOT, and -1. A character value of -1 eliminates the effect of that character. The "t_brkc" character, by default -1, acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase and kill) identical.

The calls are:

TIOCSETC
: Set the various special characters to those given in the structure.

TIOCGETC
: Fetch the special character values associated with the terminal, and store them in the pointed-to structure.

FIONREAD
: Return the number of characters currently in a terminal's input buffer into the integer pointer rg.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

getty(1M), stty (1), signal(2), ioctl(2)

NAME

a.out - assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

A.out is the output file of the assembler _as_(1) and the link loader _ld_(1). _Ld_(1) makes _a.out_ executable if there were no errors and no unresolved external references. Layout information as given in the include file for the 68000 is:

```
/*
 *    Layout of a.out file :
 *
 *    header of 8 longs magic number 405, 407, 410, 411
 *                      text size        )
 *                      data size        ) in bytes
 *                      bss size         )
 *                      symbol table size )
 *                      text relocation size    )
 *                      data relocation size    )
 *                      entry point
 *
 *    header:                 0
 *    text:             32
 *    data:             32+textsize
 *    symbol table:           32+textsize+datasize
 *    text relocation:  32+textsize+datasize+symsize
 *    data relocation:  32+textsize+datasize+symsize+rtextsize
 *
 */

/* various parameters */
#define SYMLENGTH  50                   /* maximum length of a symbol */

/* types of files */
#define     ARCMAGIC 0177545            /* ar files */
#define     FMAGIC      0407            /* standard executable */
#define     NMAGIC      0410            /* shared text executable */

/* symbol types */
#define     EXTERN      040             /* external */
#define     UNDEF       00              /* undefined */
#define     ABS         01              /* absolute */
#define     TEXT        02              /* text */
#define     DATA        03              /* data */
#define     BSS         04              /* bss */
#define     COMM        05              /* internal use only */
#define     REG         06              /* register name */
```

```
/* relocation regions */
#define     RTEXT 00
#define     RDATA 01
#define     RBSS  02
#define     REXT  03

/* relocation sizes */
#define RBYTE     00
#define RWORD     01
#define RLONG     02

/* macros which define various positions in file based on a bhdr, filhdr */
#define TEXTPOS   ((long) sizeof(filhdr))
#define DATAPOS   (TEXTPOS + filhdr.tsize)
#define SYMPOS    (DATAPOS + filhdr.dsize)
#define RTEXTPOS  (SYMPOS + filhdr.ssize)
#define RDATAPOS  (RTEXTPOS + filhdr.rtsize)
#define ENDPOS    (RDATAPOS + filhdr.rdsize)

/* header of a.out files */
struct bhdr {
      long  fmagic;
      long  tsize;
      long  dsize;
      long  bsize;
      long  ssize;
      long  rtsize;
      long  rdsize;
      long  entry;
};

/* symbol management */
struct sym {
      char  stype;              /* symbol type */
      char  sympad;             /* pad to short align */
      long  svalue;             /* value */
};

/* relocation commands */
struct reloc {
      unsigned rsegment:2;      /* RTEXT, RDATA, RBSS, or REXTERN */
      unsigned rsize:2;         /* RBYTE, RWORD, or RLONG */
      unsigned rdisp:1;         /* 1 => a displacement */
      unsigned relpad1:3;       /* pad 1 */
      char relpad2;             /* pad 2 */
      short rsymbol;            /* id of the symbol of external relocations */
      long rpos;                /* position of relocation in segment */
};
```

```
struct nlist {     /* symbol table entry */
      char         n_name[8];    /* symbol name */
      int          n_type;       /* type flag */
      unsigned     n_value;      /* value */
};

            /* values for type flag */
#define     N_UNDF      0       /* undefined */
#define     N_ABS       01      /* absolute */
#define     N_TEXT      02      /* text symbol */
#define     N_DATA      03      /* data symbol */
#define     N_BSS       04      /* bss symbol */
#define     N_TYPE      037
#define     N_REG       024     /* register name */
#define     N_FN        037     /* file name symbol */
#define     N_EXT       040     /* external bit, or'ed in */
#define     FORMAT      "%06o"  /* to print a value */
```

The file has four sections: a header, the program and data text, a symbol table, and relocation information. 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(1).

In the header the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

When an a.out file 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 data), and a stack. The text segment begins at the user program start address in the core image; the header is not loaded. If the magic number in the header is FMAGIC, 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 NMAGIC, the data segment begins at the next segment boundary following the text segment, and the text segment is not writeable by the program; if other processes are executing the same file, they will share the text segment.

The stack will occupy the highest possible user program locations in the core image and will grow downwards. The stack is automatically extended as required. The data segment is only extended as requested by brk(2).

The start of the text segment in the file is 32(10); the start of the data segment is 32+St (the size of the text) the start of the relocation information is 32+St+Sd; the start of the symbol table is 32+2(St+Sd) if the relocation information is present, 32+St+Sd if not.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file.

If a 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 information for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it will appear in the form of the structure shown above.

SEE ALSO
as(1), ld(1), nm(1)

NAME

acct - execution accounting file

SYNOPSIS

```
#include <sys/acct.h>
```

DESCRIPTION

Acct(2) causes entries to be made into an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

```
Accounting Structures

typedef unsigned short comp_t;
        /* "floating pt": 3 bits base 8 exp, 13 bits fraction */

struct  acct{

        char    ac_comm[10];    /* Accounting command name */
        comp_t  ac_utime;       /* Accounting user time */
        comp_t  ac_stime;       /* Accounting system time */
        comp_t  ac_etime;       /* Accounting elapsed time */
        time_t  ac_btime;       /* Beginning time */
        short   ac_uid;         /* Accounting user ID */
        short   ac_gid;         /* Accounting group ID */
        short   ac_mem;         /* average memory usage */
        comp_t  ac_io;          /* number of disk IO blocks */
        dev_t   ac_tty;         /* control typewriter */
        char    ac_flag;        /* Accounting flag */
};

extern  struct  acct            acctbuf;
extern  struct  inode           *acctp;/* inode of accounting file */

#define AFORK   01              /* has executed fork, but no exec */
#define ASU     02              /* used super-user privileges */
```

If the process does an exec(2), the first 10 characters of the filename appear in "ac_comm". The accounting flag contains bits indicating whether exec(2) was ever accomplished, and whether the process ever had super-user privileges.

SEE ALSO

acct(2), sa(1)

NAME
ar - archive (library) file format

SYNOPSIS
#include <ar.h>

DESCRIPTION
The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARFMAG  0177545

struct  ar_hdr {
        char    ar_name[14];
        long    ar_date;
        short   ar_uid;
        short   ar_gid;
        short   ar_mode;
        long    ar_size;
};
```

The "ar_fmag" field contains the 32-bit number ARFMAG to help verify the presence of a header. The name is a blank padded string. The other fields are left-adjusted, blank-padded numbers. They are decimal except for "ar_mode", which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on an even (0 mod 2) boundary; a new-line is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

SEE ALSO
ar(1), ld(1), nm(1)

BUGS
File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

NAME

checklist - list of file systems processed by fsck

DESCRIPTION

Checklist resides in directory /etc and contains a list of at most 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck*(1M) command.

FILES

/etc/checklist

SEE ALSO

fsck(1M)

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*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The first 2048 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.

In general the debugger *adb*(1) is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2)

NAME

dir - format of directories

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dir.h>
```

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry; see _filsys_(5). The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ    14
#endif
struct  direct {
        ino_t     d_ino;
        char      d_name[DIRSIZ];
};
```

By convention, the first two names in each directory are the names "." and "..". The first is an entry for the directory itself. By opening the file "." a program can read the names of files and subdirectories in a directory. The second name ".." is for the parent directory. The meaning of ".." is modified for the root directory of the master file system (/), where ".." has the same meaning as ".".

SEE ALSO

filsys(5)

NAME

dump, ddate - incremental dump format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ino.h>
#include <dumprestor.h>
```

DESCRIPTION

Tapes used by _dump_ and _restor_(1) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <_dumprestor.h_> is:

```
#define NTREC    10
#define MLEN     16
#define MSIZ     4096

#define TS_TAPE  1
#define TS_INODE 2
#define TS_BITS  3
#define TS_ADDR  4
#define TS_END   5
#define TS_CLRI  6
#define MAGIC    (int)60011
#define CHECKSUM (int)84446
struct           spcl
{
      int        c_type;
      time_t     c_date;
      time_t     c_ddate;
      int        c_volume;
      daddr_t    c_tapea;
      ino_t      c_inumber;
      short      c_idummy;/* pad to force long boundary */
      int        c_nrec;/* number of records per storage medium */
      int        c_magic;
      int        c_checksum;
      struct     dinodec_dinode;
      int        c_count;
      char       c_addr[BSIZE];
} spcl;

struct           idates
{
      char       id_name[16];
      char       id_incno;
```

```
        time_t      id_ddate;
};
```

NTREC is the number of 512 byte records in a physical tape block. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the "c_type" field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE Tape volume label

TS_INODE
A file or directory follows. The "c_dinode" field is a copy of the disk inode and contains bits telling what sort of file this is.

TS_BITS A bit map follows. This bit map has a one bit for each inode that was dumped.

TS_ADDR A subrecord of a file description. See "c_addr" below.

TS_END End of tape record.

TS_CLRI A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.

MAGIC All header records have this number in "c_magic".

CHECKSUM
Header records checksum to this value.

The fields of the header structure are as follows:

c_type The type of the header.

c_date The date the dump was taken.

c_ddate The date the file system was dumped from.

c_volume The current volume number of the dump.

c_tapea The current number of this (512-byte) record.

c_inumber
The number of the inode being dumped if this is of type TS_INODE.

c_magic This contains the value MAGIC above, truncated as needed.

c_checksum
This contains whatever value is needed to make the record sum to CHECKSUM.

c_dinode This is a copy of the inode as it appears on the file system; see *filsys*(5).

c_count The count of characters in "c_addr".

c_addr An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, TS_ADDR records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a *TS END* record and then the tapemark.

The structure *idates* describes an entry of the file */etc/ddate* where dump history is kept. The fields of the structure are:

id_name The dumped filesystem is '/dev/*id nam*'.
id_incno The level number of the dump tape; see *dump*(1).
id_ddate The date of the incremental dump in system format see *types*(5).

FILES
/etc/ddate

SEE ALSO
dump(1), restor(1), filsys(5), types(5)

NAME

environ - user environment

SYNOPSIS

extern char **environ;

DESCRIPTION

An array of strings called the 'environment' is made available by *exec*(2) when a process begins. By convention these strings have the form '*name*=*value*'. The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*, *time*, *nice*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by ':'.
Login(1) sets :
PATH=:/bin;/usr/bin.

HOME A user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff*, *more*, or *vi*, which may exploit special terminal capabilities. See /*etc*/*termcap* or (*termcap*(5)) for a list of terminal types.

SHELL The file name of the users login shell.

TERMCAP The string describing the terminal in TERM, or the name of the termcap file, see *termcap*(5).

EXINIT A startup list of commands read by *ex*(1), *edit*(1), and *vi*(1).

USER The login name of the user.

Further names may be placed in the environment by the *export* command and 'name=value' arguments in *sh*(1), or by the *setenv* command if you use *csh*(1). Arguments may also be placed in the environment at the point of an *exec*(2). It is unwise to conflict with certain *sh*(1) variables that are frequently exported by ".profile" files: MAIL, PS1, PS2, IFS.

SEE ALSO

csh(1), ex(1), login(1), sh(1), exec(2), system(3), termcap(5), term(7)

NAME

filsys, flblk, ino - format of file system volume

SYNOPSIS

```
#include <sys/types.h>
#include <sys/flbk.h>
#include <sys/filsys.h>
#include <sys/ino.h>
```

DESCRIPTION

Every file system storage volume (e.g floppy disk, hard disk, or tape) has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the _super block_. The layout of the super block as defined by the include file <_sys_/_filsys_._h_> is:

```
Structure of the super-block
struct      filsys {
      unsigned short s_isize; /* size in blocks of i-list */
      daddr_t     s_fsize;     /* size in blocks of entire volume */
      short       s_nfree;     /* number of addresses in s_free */
      daddr_t     s_free[NICFREE];/* free block list */
      short       s_ninode;    /* number of i-nodes in s_inode */
      ino_t       s_inode[NICINOD];/* free i-node list */
      char        s_flock;     /* lock during free list manipulation */
      char        s_ilock;     /* lock during i-list manipulation */
      char        s_fmod;      /* super block modified flag */
      char        s_ronly;     /* mounted read-only flag */
      time_t      s_time;      /* last super block update */
      daddr_t     s_tfree;     /* total free blocks*/
      ino_t       s_tinode;    /* total free inodes */
      short       s_m;         /* interleave factor */
      short       s_n;         /* " " */
      char        s_fname[6];  /* file system name */
      char        s_fpack[6];  /* file system pack name */
};
```

"S_isize" is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus i-list is s_isize -2 blocks long. "S_fsize" is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The "s_free" array contains, in "s_free[1], ... , s_free[s_nfree-1]," up to NICFREE

free block numbers. NICFREE is a configuration constant. "S_free[0]" is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file <sys/fblk.h> is:

```
struct fblk {
      short       df_nfree;
      daddr_t     df_free[NICFREE];
};
```

The fields "df_nfree" and "df_free" in a free block are used exactly like "s_nfree" and "s_free" in the super block. To allocate a block: decrement "s_nfree," and the new block number is "s_free[s_nfree]". If the new block address is 0, there are no blocks left, so give an error. If "s_nfree" became 0, read the new block into "s_nfree" and "s_free". To free a block, check if "s_nfree" is NICFREE; if so, copy "s_nfree" and the "s_free" array into it, write it out, and set "s_nfree" to 0. In any event set "s_free"[s_nfree] to the freed block's address and increment "s_nfree".

"S_ninode" is the number of free i-numbers in the s_inode array. To allocate an i-node: if "s_ninode" is greater than 0, decrement it and return s_inode[s_ninode]. If it was 0, read the i-list and place the numbers of all free inodes (up to NICINOD) into the s_inode array, then try again. To free an i-node, provided "s_ninode" is less than NICINODE, place its number into s_inode[s_ninode] and increment "s_ninode". If "s_ninode" is already NICINODE, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

"S_flock" and "s_ilock" are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of "s_fmod" on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information. "S_ronly" is a write-protection indicator; its disk value is also immaterial.

"S_time" is the last time the super-block of the file system was changed. During a reboot, "s_time" of the super-block for the root file system is used to set the system's idea of the time.

The fields "s_tfree", "s_tinode", "s_fname" and "s_fpack" are not currently maintained.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-nodes are 64 bytes long, so 8 of them fit into a block. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node as given in the include file <sys/ino.h> is:

```
Inode structure as it appears on a disk block.
struct dinode {
      unsigned short di_mode;        /* mode and type of file */
      short di_nlink;                /* number of links to file */
      short di_uid;                  /* owner's user id */
      short di_gid;                  /* owner's group id */
      off_t di_size;                 /* number of bytes in file */
      char        di_addr[40];              /* disk block addresses */
      time_t      di_atime;                 /* time last accessed */
      time_t      di_mtime;                 /* time last modified */
      time_t      di_ctime;                 /* time created */
};
#define     INOPB 8     /* 8 inodes per block */
/*
 * the 40 address bytes:
 *    39 used; 13 addresses
 *    of 3 bytes each.
 */
```

"Di_mode" tells the kind of file; it is encoded identically to the "st_mode" field of <u>stat</u>(2). "Di_nlink" is the number of directory entries (links) that refer to this i-node. "Di_uid" and "di_gid" are the owner's user and group IDs. <u>Size</u> is the number of bytes in the file. "Di_atime" and "di_mtime" are the times of last access and modification of the file contents (read, write or create) (see <u>times</u>(2)); "Di_ctime" records the time of last modification to the inode or to the file, and is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the "di_addr" field is occupied by the device code (see <u>types</u>(5)). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array "di_addr" packed into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresses are singly, doubly, and triply indirect and point to blocks of 128 block pointers. Pointers in indirect blocks have the type "daddr_t" (see <u>types</u>(5)).

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

SEE ALSO

icheck(1), dcheck(1), dir(5), mount(1), stat(2), types(5)

NAME

group - group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1), crypt(3), passwd(1), passwd(5)

NAME

mtab - mounted file system table

DESCRIPTION

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

Each entry is 64 bytes long; the first 32 bytes are the null-padded name of the place where the special file is mounted; the second 32 bytes 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*(*1*) if there are duplicated entries nor to *umount*(*1*) if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(2)

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
user's real name, and other information if desired
initial working directory
program to use as Shell *sh*(*1*) or *csh*(*1*)

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself *sh*(*1*) is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and is used, for example, by *ls*(*1*), to map numerical user ID's to names.

FILES
/etc/passwd

SEE ALSO
login(1), passwd(1)

NAME

sccsfile - format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the _checksum_, the _delta table_ (contains information about each delta), _user names_ (contains login names and/or numerical group IDs of users who may add deltas), _flags_ (contains definitions of internal keywords), _comments_ (contains arbitrary descriptive information about the file), and the _body_ (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as _the control character_ and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The @h provides a _magic number_ of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
   .
   .
   .
@c <comments> ...
   .
   .
   .
@e
```

The first line (@s) contains the number of lines inserted/deleted/unchanged respectively. The second line (@d) contains the type of the delta (currently, normal: D, and removed: R), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the @c lines contain comments associated with the delta.

The @e line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see admin(1) for more information on their use). Each flag line takes the form:

```
@f &lt;flag&gt;   &lt;optional text&gt;
```

The following flags are defined:

```
@f t  <type of program>
@f v  <program name>
@f i
@f b
@f m  <module name>
@f f  <floor>
@f c  <ceiling>
@f d  <default-sid>
@f n
@f j
@f l  <lock-releases>
@f q  <user defined>
```

The t flag defines the replacement for the %Y% identification keyword. The v flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The i flag controls the warning/error aspect of the ``No id keywords'' message. When the i flag is not present, this message is only a warning; when the i flag is present, this message will cause a ``fatal'' error (the file will not be gotten, or the delta will not be made). When the b flag is present the -b keyletter may be used on the _get_ command to cause a branch in the delta tree. The m flag defines the first choice for the replacement text of the %M% identification keyword. The f flag defines the ``floor'' release; the release below which no deltas may be added. The c flag defines the ``ceiling'' release; the release above which no deltas may be added. The d flag defines the default SID to be used when none is specified on a _get_ command. The n flag causes _delta_ to insert a ``null'' delta (a delta that applies _no_ changes) in those releases that are skipped when a delta is made in a _new_ release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the n flag causes skipped releases to be completely empty. The j flag causes _get_ to allow concurrent edits of the same base SID. The l flag defines a _list_ of releases that are _locked_ against editing (_get_(1) with the -e keyletter). The q flag defines the replacement for the %Q% identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines @t and @T. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: _insert_, _delete_, and _end_, represented by:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).
Source Code Control System User's Guide by L. E. Bonanni and C. A. Salemi.

NAME

termcap - terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Entries in *termcap* consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed

dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give :ei=: if ic
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no cm)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing is
im	bool		Insert mode (enter); give :im=: if ic
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by other function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of keypad transmit mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of other keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in keypad transmit mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on other function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no cm)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM2500,H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with is)
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)

tc	str	Entry of similar terminal - must be last
te	str	String to end programs that use cm
ti	str	String to begin programs that use cm
uc	str	Underscore one char and move past it
ue	str	End underscore mode
ug	num	Number of blank chars left by us or ue
ul	bool	Terminal underlines even though it doesn't overstrike
up	str	Upline (cursor up)
us	str	Start underscore mode
vb	str	Visible bell (may not move cursor)
ve	str	Sequence to end open/visual mode
vs	str	Sequence to start open/visual mode
xb	bool	Beehive (f1=escape, f2=ctrl C)
xn	bool	A newline is ignored after a wrap (Concept)
xr	bool	Return acts like ce \r \n (Delta Data)
xs	bool	Standout not erased by writing over it (HP 264?)
xt	bool	Tabs are destructive, magic so char (Teleray 1061)

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the termcap file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
c1|c100|concept100:is=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200:\
        :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
        :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:mi:nd=\E=
        :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in termcap are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has automatic margins (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability am. Hence the description of the Concept includes am. Numeric capabilities are followed by the character '#' and then the value. Thus co which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as ce (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form "3.5" to specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A \E maps to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences \n \r \t \b \f give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a : in a capability it must be escaped in octal as \072. If it is necessary to place a null character in a string capability it must be encoded as \200. The routines which deal with _termcap_ use C strings, and strip the high bits of the output very late so that a \200 comes out as a \000 would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in _termcap_ and to build up a description gradually, using partial descriptions with _ex_ to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the _termcap_ file to describe it or bugs in _ex_. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in _/etc/termcap_. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor.

Basic capabilities

The number of columns on each line for the terminal is given by the co numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the li capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the am capability. If the terminal can clear its screen, then this is given by the cl string capability. If the terminal can backspace, then it should have the bs capability, unless a backspace is accomplished by a character other than ^H (ugh) in which case you should give this character as the bc string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the os capability.

A very important point here is that the local cursor motions encoded in _termcap_ are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the am capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the termcap file usually assumes that this is on, i.e. am.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the model 33 teletype is described as

    t3|33|tty33:co#72:os

while the Lear Siegler ADM-3 is described as

    cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80

Cursor addressing

Cursor addressing in the terminal is described by a cm string capability, with printf(3s) like escapes %x in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the cm string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the % encodings have the following meanings:

%d	as in printf, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds x to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.
%D	Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its cm capability is cm=6\E&a%r%2c%2Y. The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, cm=^T%.%.. Terminals which use %. need to be able to backspace the cursor (bs or bc), and to move the cursor up one line on the screen (up introduced below). This is necessary because it is not always safe to transmit \t, \n ^D and \r, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cm=\E=%+ %+ .

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as nd (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as up. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as ho; similarly a fast way of getting to the lower left hand corner can be given as ll; this may involve going up with up from the home position, but the editor will never do this itself (unless ll does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as ce. If the terminal can clear from the current position to the end of the display, then this should be given as cd. The editor only uses cd from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as al; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as dl; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as sb, but just al suffices. If the terminal can retain display memory above then the da capability should be given; if display memory can be retained below then db should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with sb may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using _termcap_. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc   def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability in, which stands for insert null. If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as im the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as ei the sequence to leave insert mode (give this, with an empty value also if you gave im so). Now give as ic any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give ic, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in ip (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in ip.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability mi to speed up inserting in this case. Omitting mi will affect only speed. Some terminals (notably Datamedia's) must not have mi because of the way their insert mode works.

Finally, you can specify delete mode by giving dm and ed to enter and exit delete mode, and dc to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as so and se respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining - half bright is not usually an acceptable standout mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then ug should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as us and ue respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as uc. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

Many terminals, such as the HP 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as vb; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of _ex_, this can be given as vs and ve, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as ti and te. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability ul. If overstrikes are erasable with a blank, then this should be indicated by giving eo.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as ks and ke. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as kl, kr, ku, kd, and kh respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as k0, k1, ..., k9. If these keys have labels other than the default f0 through f9, the labels can be given as l0, l1, ..., l9. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the _termcap_ 2 letter codes can be given in the ko capability, for example, :ko=cl,ll,sf,sb:, which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The ma entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with kl, kr, ku, kd, and kh. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are h for kl, j for kd, k for ku, l for kr, and H for kh. For example, the mime would be :ma=^Kj^Zk^Xl: indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as pc.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, then this can be given as ta.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate hz. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate nc. Early Concept terminals, which ignore a linefeed immediately after an am wrap, should indicate xn. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), xs should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate xt. Other specific terminal problems may be corrected by adding more capabilities of the form x<u>x</u>.

Other capabilities include is, an initialization string for the terminal, and if, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, is will be printed before if. This is useful where if is /<u>usr</u>/<u>lib</u>/<u>tabset</u>/<u>std</u> but is clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability tc can be given with the name of the similar terminal. This capability must be <u>last</u> and the combined length of the two entries must not exceed 1024. Since <u>termlib</u> routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with xx@ where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the ks or ke capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3), termcap(3), tset(1), vi(1), ul(1), more(1)

AUTHOR

William Joy
Mark Horton added underlining and keypad support

BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap*(*3*) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The ma, vs, and ve entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

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 *bproc*(8).

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:

```
struct {
        char  pathname[32];
        int   mode;
        char  uid;
        char  gid;
        char  unused1;
        char  size[3];
        long  modtime;
        int   tapeaddr;
        char  unused2[16];
        int   checksum;
};
```

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 (see file system *filsys*(5)). 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 above 25 (resp. 63) are available for file storage.

A fake entry has a size of zero.

SEE ALSO

filsys(5), tp(1)

BUGS

The *pathname*, *uid*, *gid*, and *size* fields are too small.

NAME

ttys - terminal initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective.

The second character is used as an argument to *getty*(1M), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*.

The following chart lists the characters to be used for the second character:

Single	*Speed*
1	50
2	75
3	110
4	134.5
5	150
6	200
7	300
8	600
9	1200
a	1800
b	2400
c	4800
d	9600
e	Ext A and 19200
f	Ext B

CONSOLES

A	110 console
B	Decwriter
C	Interdata

OTHERS

D-E-F-G	300/1200/150/110 for modems
H-I	1200/300 for modems

The remainder of the line is the terminal's entry in the device directory, /dev.

EXAMPLE

```
1dconsole
1dtty0
19tty1
19tty2
009tty3
07ttyd0
```

FILES

/etc/ttys

SEE ALSO

init(1M), getty(1M), login(1)

NAME

ttytype - data base of terminal types by port

DESCRIPTION

Ttytype is a database containing, for each tty port on the system, the kind of terminal that is attached to it. There is one line per port, containing the terminal kind (as a name listed in termcap (5)), a space, and the name of the tty, minus /dev/.

This information is read by *tset*(1) and by *login*(1) to initialize the TERM environment variable at login time.

EXAMPLE

```
dw console
3a tty0
h19 tty1
h19 tty2
du ttyd0
```

FILES

/etc/ttytype

SEE ALSO

tset(1), login(1)

NAME

types - primitive system data types

SYNOPSIS

```
#include <sys/types.h>
```

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
typedef     long        daddr_t;
typedef     char *      caddr_t;
typedef     long        mem_t;
typedef     unsigned short    ino_t;
typedef     long        time_t;
typedef     long        label_t[13];      /* regs d2-d7, a2-a7, pc */
typedef     short       dev_t;
typedef     long        off_t;

       /* selectors and constructor for device code */

#define     major(x)    (int)(((unsigned)(x) >> 8))
#define     minor(x)    (int)(x) & 0377)
#define     makedev(x,y)      (dev_t)((x) << 8|(y))
```

The form _daddr t_ is used for disk addresses except in an i-node on disk, see _filsys_(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The _label t_ variables are used to save the processor state while another process is running.

SEE ALSO

filsys(5), time(2), lseek(2), adb(1)

NAME

utmp, wtmp - login records

SYNOPSIS

```
#include <utmp.h>
```

DESCRIPTION

The *utmp* file allows one to discover information about who is currently using the system. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
        char  ut_line[8];        /* tty name */
        char  ut_name[8];        /* user id */
        long  ut_time;           /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(2).

The *wtmp* file records all logins and logouts. Its format is exactly like *utmp* except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name "~" indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names "|" and "}" indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by *login*(1) and *init*(1M). Neither of these programs creates the file, so if it is removed record-keeping is turned off.

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

login(1), init(1M), who(1)

NAME

wtmp - user login history

DESCRIPTION

This file records all logins and logouts. Its format is exactly like _utmp_(5) 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_(1) and _init_(1M). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by _ac_(1).

FILES

/usr/adm/wtmp

SEE ALSO

utmp(5), login(1), init(1M), who(1)

NAME

adventure - an exploration game

SYNOPSIS

/usr/games/adventure

DESCRIPTION

The object of the game is to locate and explore Colossal Cave, find the treasures hidden there, and bring them back to the building with you. The program is self-describing to a point, but part of the game is to discover its rules.

To terminate a game, type "quit"; to save a game for later resumption, type "suspend".

BUGS

Saving a game creates a large executable file instead of just the information needed to resume the game.

NAME

aliens - The alien invaders attack the earth

SYNIOPSIS

/usr/games/aliens

DESCRIPTION

This is a UNIX version of Space Invaders. The program is pretty much self documenting.

FILES

/usr/games/lib/aliens.log Score file

BUGS

The program is a CPU hog. It needs to be re-written. It doesn't do well on terminals that run slower than 9600 baud.

NAME

arithmetic - provide drill in number facts

SYNOPSIS

/usr/games/arithmetic [+-x/] [range]

DESCRIPTION

Arithmetic types out simple arithmetic problems, and waits for an answer to be typed in. If the answer is correct, it types back Right!, and a new problem. If the answer is wrong, it replies What?, and waits for another answer. Every twenty problems, it publishes statistics on correctness and the time required to answer.

To quit the program, type an interrupt (delete).

The first optional argument determines the kind of problem to be generated; +-x/ respectively cause addition, subtraction, multiplication, and division problems to be generated. One or more characters can be given; if more than one is given, the different types of problems will be mixed in random order; default is +-

Range is a decimal number; all addends, subtrahends, differences, multiplicands, divisors, and quotients will be less than or equal to the value of _range_. Default _range_ is 10.

At the start, all numbers less than or equal to _range_ are equally likely to appear. If the respondent makes a mistake, the numbers in the problem which was missed become more likely to reappear.

As a matter of educational philosophy, the program will not give correct answers, since the learner should, in principle, be able to calculate them. Thus the program is intended to provide drill for someone just past the first learning stage, not to teach number facts _de novo_. For almost all users, the relevant statistic should be time per problem, not percent correct.

NAME

backgammon - the game

SYNOPSIS

/usr/games/backgammon

DESCRIPTION

This program does what you expect. It will ask whether you need instructions.

NAME

banner - print large banner on printer

SYNOPSIS

/usr/games/banner [-w*n*] message ...

DESCRIPTION

Banner prints a large, high quality banner on the standard output. If the message is omitted, it prompts for and reads one line of its standard input. If -w is given, the output is scrunched down from a width of 132 to *n* , suitable for a narrow terminal. If *n* is omitted, it defaults to 80.

The output should be printed on a hard-copy device, up to 132 columns wide, with no breaks between the pages. The volume is enough that you want a printer or a fast hardcopy terminal, but if you are patient, a decwriter or other 300 baud terminal will do.

BUGS

Several ASCII characters are not defined, notably <, >, [,], \, ^, _, {, }, |, and ~. Also, the characters ", ', and & are funny looking (but in a useful way.)

The -w option is implemented by skipping some rows and columns. The smaller it gets, the grainier the output. Sometimes it runs letters together.

AUTHOR

Mark Horton

NAME

bcd - convert to antique media

SYNOPSIS

/usr/games/bcd text

DESCRIPTION

Bcd converts the literal _text_ into a form familiar to old-timers.

NAME

fish - play ``Go Fish''

SYNOPSIS

/usr/games/fish

DESCRIPTION

Fish plays the game of Go Fish, a childrens' card game. The Object is to accumulate `books' of 4 cards with the same face value. The players alternate turns; each turn begins with one player selecting a card from his hand, and asking the other player for all cards of that face value. If the other player has one or more cards of that face value in his hand, he gives them to the first player, and the first player makes another request. Eventually, the first player asks for a card which is not in the second player's hand: he replies `GO FISH!' The first player then draws a card from the `pool' of undealt cards. If this is the card he had last requested, he draws again. When a book is made, either through drawing or requesting, the cards are laid down and no further action takes place with that face value.

To play the computer, simply make guesses by typing a, 2, 3, 4, 5, 6, 7, 8, 9, 10, j, q, or k when asked. Hitting return gives you information about the size of my hand and the pool, and tells you about my books. Saying `p' as a first guess puts you into `pro' level; The default is pretty dumb.

NAME

fortune - print a random, hopefully interesting, adage

SYNOPSIS

fortune [-wsl]

DESCRIPTION

Fortune with no arguments prints out a random adage. The flags mean:

-w Waits before termination for an amount of time calculated from the number of characters in the message. This is useful if it is executed as part of the logout procedure to guarantee that the message can be read before the screen is cleared.

-s Short messages only.

-l Long messages only.

FILES

/usr/games/lib/fortunes.dat

AUTHOR

Ken Arnold

NAME

hangman - Computer version of the game hangman

SYNOPSIS

/usr/games/hangman

DESCRIPTION

In _hangman_, the computer picks a word from the on-line word list and you must try to guess it. The computer keeps track of which letters have been guessed and how many wrong guesses you have made on the screen.

FILES

/usr/dict/words On-line word list

NAME
life - play the game of life

SYNOPSIS
life [-r]

DESCRIPTION
Life is a pattern generating game set up for interactive use on a video terminal. The way it operates is: You use a series of commands to set up a pattern on the screen then let it generate further patterns from that pattern.

The algorithm used is: For each square in the matrix, look at it and its eight adjacent neighbors. If the present square is not occupied and exactly three of its neighbor squares are occupied, then that square will be occupied in the next pattern. If the present square is occupied and two or three of its neighbor squares are occupied, then that square will be occupied in the next pattern. Otherwise, the present square will not be occupied in the next pattern.

The edges of the screen are normally treated as an unoccupied void. If you specify the -_r_ option on the command line, the screen is treated as a sphere; that is, the top and bottom lines are considered adjacent and the left and right columns are considered adjacent.

The pattern generation number and the number of occupied squares are displayed in the lower left hand corner.

Below is a list of commands available to the user. A # stands for any number. A ^ followed by a capital letter represents a control character.

#,#a Add a block of elements. The first number specifies the horizontal width. The second number specifies the vertical width. If a number is not specified, the default is 1.

#c Step through the next # patterns. If no number is specified, step forever. The operation can be aborted by typing rubout (delete).

#,#d Delete a block of elements. The first number specifies the horizontal width. The second number specifies the vertical width. If a number is not specified, the default is 1.

#f Generate a little flier at the present location. The number (modulo 8) determines the direction.

#,#g Move to absolute screen location. The first number specifies the horizontal location. The second number specifies the vertical location. If a number is not specified, the default is 0.

#h Move left # steps. If no number is specified, the default is 1.

#j Move down # steps. The default is 1.

#k Move up # steps. The default is 1.

#l Move right # steps. The default is 1.

#n Step through the next # patterns. If no number is specified, generate the next pattern. The operation can be aborted by typing rubout (delete).

p Put the last yanked or deleted block at the present location.

q Quit.

#,#y Yank a block of elements. The first number specifies the horizontal width. The second number specifies the vertical width. If a number is not specified, the default is 1.

C Clear the pattern.

#F Generate a big flier at the present location. The number (modulo 8) determines the direction.

#H Move to the left margin.

#J Move to the bottom margin.

#K Move to the top margin.

#L Move to the right margin.

#^H Move left # steps. If no number is specified, the default is 1.

#^J Move down # steps. The default is 1.

#^K Move up # steps. The default is 1.

#^L Move right # steps. The default is 1.

^R Redraw the screen. This is used for those occasions when the terminal screws up.

. Repeat the last add (a) or delete (d) operation.

; Repeat the last move (h, j, k, l) operation.

AUTHOR

Asa Romberger

BUGS

The following features are planned but not implemented:

#,#S	Save the selected area in a file.
R	Restore from a file.
m	Generate a macro command.
!	Shell escape.
e	Edit a file.
i	Input commands from a file.

NAME

number - convert Arabic numerals to English

SYNOPSIS

/usr/games/number

DESCRIPTION

Number copies the standard input to the standard output, changing each decimal number to a fully spelled out version.

NAME

rain - animated raindrops display

SYNOPSIS

rain

DESCRIPTION

Rain's display is modeled after the VAX/VMS program of the same name. The terminal has to be set for 9600 baud to obtain the proper effect.

As with all programs that use *termcap*, the TERM environment variable must be set (and exported) to the type of the terminal being used.

FILES

/etc/termcap

AUTHOR

Eric P. Scott

NAME

trek - trekkie game

SYNOPSIS

/usr/games/trek [[-a] file]

DESCRIPTION

Trek is a game of space glory and war. Below is a summary of commands. For complete documentation, see Trek by Eric Allman.

If a filename is given, a log of the game is written onto that file. If the -a flag is given before the filename, that file is appended to, not truncated.

The game will ask you what length game you would like. Valid responses are short, medium, and long. You may also type restart, which restarts a previously saved game. You will then be prompted for the skill, to which you must respond novice, fair, good, expert, commadore, or impossible. You should normally start out with a novice and work up.

In general, throughout the game, if you forget what is appropriate the game will tell you what it expects if you just type in a question mark.

AUTHOR

Eric Allman

SEE ALSO

/usr/doc/trek

COMMAND SUMMARY

abandon	capture
cloak up/down	
computer request; ...	damages
destruct	dock
help	impulse course distance
lrscan	move course distance
phasers automatic amount	
phasers manual amtl coursel spreadl ...	
torpedo course [yes] angle/no	
ram course distance	rest time
shell	shields up/down
srscan [yes/no]	
status	terminate yes/no
undock	visual course
warp warp_factor	

NAME

twinkle - twinkle stars on the screen

SYNOPSIS

/usr/games/twinkle [-+[s save]] [density1] [density2]

DESCRIPTION

Twinkle causes a specified density of "stars" to twinkle on the screen. The following options are available;

- `-` print out the present screen density (the percentage of the screen that will be filled with stars) in the lower left hand corner of the screen. This number will change as stars go on and off.

- `+` do not "randomize" before starting. The screen starts out completely blank and stars are added, bit by bit. In this case the density rises beyond the specified density, then falls to the required percentage.

- `s` save binary density on file "save", in case you want to see the density curve that a particulr density specification produced during the life of the show.

density

If no density is specified, density is .5 (50% of the screen will be filled with stars).
If only *density1* is given, density is 1/density1
If both density1 and density2 are given, *density* is the resultant of density1/density1+density2.

EXAMPLE

```
twinkle -+ 2 6
```

would start from a blank screen and twinkle stars to a final density of 2/8, or 25%. The densities would be shown in the lower left hand corner, as a three-place decimal.

AUTHOR

Asa Romberger

NAME

worm - Play the growing worm game

SYNOPSIS

worm [_size_]

DESCRIPTION

In _worm_, you are a little worm, your body is the "o"'s on the screen and your head is the "@". You move with the hjkl keys (as in the game snake). If you don't press any keys, you continue in the direction you last moved. The upper case HJKL keys move you as if you had pressed several (9 for HL and 5 for JK) of the corrosponding lower case key (unless you run into a digit, then it stops).

On the screen you will see a digit; if your worm eats the digit, it will grow longer. The actual amount by which the worm will grow longer depends upon which digit was eaten. The object of the game is to see how long you can make the worm grow.

The game ends when the worm runs into either the sides of the screen, or itself. The current score (how much the worm has grown) is kept in the upper left corner of the screen.

The optional argument, if present, is the initial length of the worm.

BUGS

If the initial length of the worm is set to less than one or more than 75, various strange things happen.

NAME

worms - animate worms on a display terminal

SYNOPSIS

worms [-field] [-length #] [-number #] [-trail]

DESCRIPTION

-field makes a "field" for the worm(s) to eat; -trail causes each worm to leave a trail behind it. You can figure out the rest by yourself.

FILES

/etc/termcap

AUTHOR

Eric P. Scott

DIAGNOSTICS

Invalid length

Value not in range 2 <= length <= 1024

Invalid number of worms

Value not in range 1 <= number <= 40

TERM: parameter not set

The TERM environment variable is not defined. Do

```
TERM=terminal type
export TERM
```

Unknown terminal type

Your terminal type (as determined from the TERM environment variable) is not defined in /etc/termcap.

Terminal not capable of cursor motion

Your terminal is too stupid to run this program.

Out of memory

This should never happen.

BUGS

The lower-right-hand character position will not be updated properly on a terminal that wraps at the right margin.

Terminal initialization is not performed.

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).

NAME

ascii - map of ASCII character set

SYNOPSIS

cat /usr/man/man7/ascii.7

DESCRIPTION

<u>Ascii</u> is a map of the ASCII character set, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041  ! |042  " |043  # |044  $ |045  % |046  & |047  ' |
|050  ( |051  ) |052  * |053  + |054  , |055  - |056  . |057  / |
|060  0 |061  1 |062  2 |063  3 |064  4 |065  5 |066  6 |067  7 |
|070  8 |071  9 |072  : |073  ; |074  < |075  = |076  > |077  ? |
|100  @ |101  A |102  B |103  C |104  D |105  E |106  F |107  G |
|110  H |111  I |112  J |113  K |114  L |115  M |116  N |117  O |
|120  P |121  Q |122  R |123  S |124  T |125  U |126  V |127  W |
|130  X |131  Y |132  Z |133  [ |134  \ |135  ] |136  ^ |137  _ |
|140  ` |141  a |142  b |143  c |144  d |145  e |146  f |147  g |
|150  h |151  i |152  j |153  k |154  l |155  m |156  n |157  o |
|160  p |161  q |162  r |163  s |164  t |165  u |166  v |167  w |
|170  x |171  y |172  z |173  { |174  | |175  } |176  ~ |177 del|

| 00 nul| 01 soh| 02 stx| 03 etx| 04 eot| 05 enq| 06 ack| 07 bel|
| 08 bs | 09 ht | 0a nl | 0b vt | 0c np | 0d cr | 0e so | 0f si |
| 10 dle| 11 dc1| 12 dc2| 13 dc3| 14 dc4| 15 nak| 16 syn| 17 etb|
| 18 can| 19 em | 1a sub| 1b esc| 1c fs | 1d gs | 1e rs | 1f us |
| 20 sp | 21  ! | 22  " | 23  # | 24  $ | 25  % | 26  & | 27  ' |
| 28  ( | 29  ) | 2a  * | 2b  + | 2c  , | 2d  - | 2e  . | 2f  / |
| 30  0 | 31  1 | 32  2 | 33  3 | 34  4 | 35  5 | 36  6 | 37  7 |
| 38  8 | 39  9 | 3a  : | 3b  ; | 3c  < | 3d  = | 3e  > | 3f  ? |
| 40  @ | 41  A | 42  B | 43  C | 44  D | 45  E | 46  F | 47  G |
| 48  H | 49  I | 4a  J | 4b  K | 4c  L | 4d  M | 4e  N | 4f  O |
| 50  P | 51  Q | 52  R | 53  S | 54  T | 55  U | 56  V | 57  W |
| 58  X | 59  Y | 5a  Z | 5b  [ | 5c  \ | 5d  ] | 5e  ^ | 5f  _ |
| 60  ` | 61  a | 62  b | 63  c | 64  d | 65  e | 66  f | 67  g |
| 68  h | 69  i | 6a  j | 6b  k | 6c  l | 6d  m | 6e  n | 6f  o |
| 70  p | 71  q | 72  r | 73  s | 74  t | 75  u | 76  v | 77  w |
| 78  x | 79  y | 7a  z | 7b  { | 7c  | | 7d  } | 7e  ~ | 7f del|
```

NAME

eqnchar - special character definitions for eqn

SYNOPSIS

eqn /usr/pub/eqnchar [files] | troff [options]

neqn /usr/pub/eqnchar [files] | nroff [options]

DESCRIPTION

Eqnchar contains _troff_ and _nroff_ character definitions for constructing characters that are not available on the Graphic Systems typesetter. These definitions are primarily intended for use with _eqn_ and _neqn_. It contains definitions for the following characters

<table>
<tr><td>"ciplus"</td><td>ciplus</td><td>"||"</td><td>||</td><td>"square"</td><td>square</td></tr>
<tr><td>"citimes"</td><td>citimes</td><td>"langle"</td><td>langle</td><td>"circle"</td><td>circle</td></tr>
<tr><td>"wig"</td><td>wig</td><td>"rangle"</td><td>rangle</td><td>"blot"</td><td>blot</td></tr>
<tr><td>"-wig"</td><td>-wig</td><td>"hbar"</td><td>hbar</td><td>"bullet"</td><td>bullet</td></tr>
<tr><td>">wig"</td><td>>wig</td><td>"ppd"</td><td>ppd</td><td>"prop"</td><td>prop</td></tr>
<tr><td>"<wig"</td><td><wig</td><td>"<->"</td><td><-></td><td>"empty"</td><td>empty</td></tr>
<tr><td>"=wig"</td><td>=wig</td><td>"<=>"</td><td><=></td><td>"member"</td><td>member</td></tr>
<tr><td>"star"</td><td>star</td><td>"|<"</td><td>|<</td><td>"nomem"</td><td>nomem</td></tr>
<tr><td>"bigstar"</td><td>bigstar</td><td>"|>"</td><td>|></td><td>"cup"</td><td>cup</td></tr>
<tr><td>"=dot"</td><td>=dot</td><td>"ang"</td><td>ang</td><td>"cap"</td><td>cap</td></tr>
<tr><td>"orsign"</td><td>orsign</td><td>"rang"</td><td>rang</td><td>"incl"</td><td>incl</td></tr>
<tr><td>"andsign"</td><td>andsign</td><td>"3dot"</td><td>3dot</td><td>"subset"</td><td>subset</td></tr>
<tr><td>"=del"</td><td>=del</td><td>"thf"</td><td>thf</td><td>"supset"</td><td>supset</td></tr>
<tr><td>"oppA"</td><td>oppA</td><td>"quarter"</td><td>quarter</td><td>"!subset"</td><td>!subset</td></tr>
<tr><td>"oppE"</td><td>oppE</td><td>"3quarter"</td><td></td><td>3quarter</td><td>"!supset"!supset</td></tr>
<tr><td>"angstrom"</td><td></td><td>angstrom</td><td>"degree"</td><td>degree</td><td></td></tr>
</table>

FILES

/usr/pub/eqnchar

SEE ALSO

troff(1), eqn(1)

NAME

greek - graphics for extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek [| greek -Tterminal]

DESCRIPTION

Greek gives the mapping from ascii to the 'shift out' graphics in effect between SO and SI on model 37 Teletypes with a 128-character type-box. These are the default greek characters produced by _nroff_. The filters of _greek_(1) attempt to print them on various other terminals. The file contains:

alpha	a	A	beta	b	B	gamma	g	\
GAMMA	G	G	delta	d	D	DELTA	D	W
epsilon	e	S	zeta	z	Q	eta	y	N
THETA	H	T	theta	h	O	lambda	l	L
LAMBDA	L	E	mu	m	M	nu	n	@
xi	c	X	pi	p	J	PI	P	P
rho	r	K	sigma	s	Y	SIGMA	S	R
tau	t	I	phi	f	U	PHI	F	F
psi	q	V	PSI	Q	H	omega	w	C
OMEGA	W	Z	nabla		[	not		_
partial		]	integral		^			

SEE ALSO

greek(1)
troff(1)

NAME
man - macros to typeset manual

SYNOPSIS
nroff -man file ...

troff -man file ...

DESCRIPTION
These macros are used to lay out pages of this manual.

The definition of these macros may be found in /usr/lib/tmac/tmac.an.

Some special features of this set of macros:

Any text argument _t_ may be zero to six words. Quotes may be used to include blanks in a ´word´. If _text_ is empty, the special treatment is applied to the next input line with text to be printed. In this way .I may be used to italicize a whole line, or .SM followed by .B to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents _i_ are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by -man:

*R _troff_.

*S Change to default type size.

EXAMPLE

    nroff -man man.7

to _nroff_ this manual section.

FILES
/usr/lib/tmac/tmac.an

SEE ALSO
man(1), troff(1)

BUGS
Relative indents don´t nest.

REQUESTS

Request	Cause Break	If no Argument	Explanation
.B *t*	no	*t*=n.t.l.*	Text *t* is bold.
.BI *t*	no	*t*=n.t.l.	Join words of *t* alternating bold and italic.
.BR *t*	no	*t*=n.t.l.	Join words of *t* alternating bold and Roman.
.DT	no	.5i 1i...	Restore default tabs.
.HP *i*	yes	*i*=p.i.*	Set prevailing indent to *i*. Begin paragraph with hanging indent.
.I *t*	no	*t*=n.t.l.	Text *t* is italic.
.IB *t*	no	*t*=n.t.l.	Join words of *t* alternating italic and bold.
.IP *x* *i*	yes	*x*=""	Same as .TP with tag *x*.
.IR *t*	no	*t*=n.t.l.	Join words of *t* alternating italic and Roman.
.LP	yes	-	Same as .PP.
.PD *d*	no	*d*=.4v	Interparagraph distance is *d*.
.PP	yes	-	Begin paragraph. Set prevailing indent to .5i.
.RE	yes	-	End of relative indent. Set prevailing indent to amount of starting .RS.
.RB *t*	no	*t*=n.t.l.	Join words of *t* alternating Roman and bold.
.RI *t*	no	*t*=n.t.l.	Join words of *t* alternating Roman and italic.
.RS *i*	yes	*i*=p.i.	Start relative indent, move left margin in distance *i*. Set prevailing indent to .5i for nested indents.
.SH *t*	yes	*t*=n.t.l.	Subhead.
.SM *t*	no	*t*=n.t.l.	Text *t* is small.
.TH *n* *c* *x*	yes	-	Begin page named *n* of chapter *c*; *x* is extra commentary, e.g. 'local', for page foot. Set prevailing indent and tabs to .5i.
.TP *i*	yes	*i*=p.i.	Set prevailing indent to *i*. Begin indented paragraph with hanging tag given by next text line. If tag doesn't fit, place it on separate line.

* n.t.l. = next text line; p.i. = prevailing indent

NAME

me - macros for formatting papers

SYNOPSIS

nroff -me [options] file ...
troff -me [options] file ...

DESCRIPTION

This package of _nroff_ and _troff_ macro definitions provides a canned formatting facility for technical papers in various formats. When producing 2-column output on a terminal, filter the output through _col_(1).

The macro requests are defined below. Many _nroff_ and _troff_ requests are unsafe in conjunction with this package, however these requests may be used with impunity after the first .pp:

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin
.ce n	center next n lines
.ul n	underline next n lines
.sz +n	add n to point size

Output of the _eqn_, _neqn_, _refer_, and _tbl_(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.e
/usr/lib/me/*

SEE ALSO

eqn(1), troff(1), refer(1), tbl(1)
-me Reference Manual, Eric P. Allman
Writing Papers with Nroff Using -me

REQUESTS

In the following list, initialization refers to the first .pp, .lp, .ip, .np, .sh, or .uh macro. This list is incomplete; see _The -me Reference Manual_ for interesting details.

Request	Initial Value	Cause Break	Explanation
.(c	-	yes	Begin centered block
.(d	-	no	Begin delayed text
.(f	-	no	Begin footnote
.(l	-	yes	Begin list
.(q	-	yes	Begin major quote
.(x _x_	-	no	Begin indexed item in index _x_
.(z	-	no	Begin floating keep
.)c	-	yes	End centered block

.)d	-	yes	End delayed text
.)f	-	yes	End footnote
.)l	-	yes	End list
.)q	-	yes	End major quote
.)x	-	yes	End index item
.)z	-	yes	End floating keep
.++ m H	-	no	Define paper section. m defines the part of the paper, and can be C (chapter), A (appendix), P (preliminary, e.g., abstract, table of contents, etc.), B (bibliography), RC (chapters renumbered from page one each chapter), or RA (appendix renumbered from page one).
.+c T	-	yes	Begin chapter (or appendix, etc., as set by .++). T is the chapter title.
.1c	1	yes	One column format on a new page.
.2c	1	yes	Two column format.
.EN	-	yes	Space after equation produced by eqn or neqn.
.EQ x y	-	yes	Precede equation; break out and add space. Equation number is y. The optional argument x may be I to indent equation (default), L to left-adjust the equation, or C to center the equation.
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TS x	-	yes	Begin table; if x is H table has repeated heading.
.ac A N	-	no	Set up for ACM style output. A is the Author's name(s), N is the total number of pages. Must be given before the first initialization.
.b x	no	no	Print x in boldface; if no argument switch to boldface.
.ba +n	0	yes	Augments the base indent by n. This indent is used to set the indent on regular text (like paragraphs).
.bc	no	yes	Begin new column
.bi x	no	no	Print x in bold italics (nofill only)
.bx x	no	no	Print x in a box (nofill only).
.ef ´x´y´z´	´´´´	no	Set even footer to x y z
.eh ´x´y´z´	´´´´	no	Set even header to x y z
.fo ´x´y´z´	´´´´	no	Set footer to x y z
.hx	-	no	Supress headers and footers on next page.
.he ´x´y´z´	´´´´	no	Set header to x y z
.hl	-	yes	Draw a horizontal line
.i x	no	no	Italicize x; if x missing, italic text follows.
.ip x y	no	yes	Start indented paragraph, with hanging tag x. Indentation is y ens (default 5).
.lp	yes	yes	Start left-blocked paragraph.
.lo	-	no	Read in a file of local macros of the form .*x. Must be given before initialization.
.np	1	yes	Start numbered paragraph.
.of ´x´y´z´	´´´´	no	Set odd footer to x y z
.oh ´x´y´z´	´´´´	no	Set odd header to x y z
.pd	-	yes	Print delayed text.
.pp	no	yes	Begin paragraph. First line indented.

.r	yes	no	Roman text follows.
.re	-	no	Reset tabs to default values.
.sc	no	no	Read in a file of special characters and diacritical marks. Must be given before initialization.
.sh _n_ _x_	-	yes	Section head follows, font automatically bold. _n_ is level of section, _x_ is title of section.
.sk	no	no	Leave the next page blank. Only one page is remembered ahead.
.sz +_n_	10p	no	Augment the point size by _n_ points.
.th	no	no	Produce the paper in thesis format. Must be given before initialization.
.tp	no	yes	Begin title page.
.u _x_	-	no	Underline argument (even in _troff_). (Nofill only).
.uh	-	yes	Like .sh but unnumbered.
.xp _x_	-	no	Print index _x_.

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 in various formats. When producing 2-column output on a terminal, filter the output through _col_(1).

EXAMPLE

```
nroff -ms -o3- filea | col
```

will _nroff_ the file starting with page 3 and produce two column output where the file contains the ''.2C'' macro. Any of the _nroff_ or _troff_ options may be used in conjunction with the -_ms_ macro package, and several files may be _nroffed_ at once.

The macro requests are defined below. Many _nroff_ and _troff_ requests may not work as expected in conjunction with this macro package. However, the following requests may be used with impunity after the first .PP:

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

Output of the _eqn_, _neqn_, and _tbl_(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.s

SEE ALSO

eqn(1), tbl(1), troff(1)
and ''Typing Documents on the System'' by M.E. Lesk.

REQUESTS

Request	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AT	no	yes	Print 'Attached' and turn off line filling.
.AU _x_ _y_	no	yes	Author's name follows. _x_ is location and _y_ is extension, ignored except in TM.

.B x	no	no	Print x in boldface; if no argument switch to boldface.
.B1	no	yes	Begin text to be enclosed in a box.
.B2	no	yes	End text to be boxed . print it.
.BT	date	no	Bottom title, automatically invoked at foot of page. May be redefined.
.BX x	no	no	Print x in a box.
.CS x...	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.CT	no	yes	Print 'Copies to' and enter no-fill mode.
.DA x	nroff	no	'Date line' at bottom of page is x. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS x	no	yes	Start of displayed text, to appear verbatim line-by-line. x=I for indented display (default), x=L for left-justified on the page, x=C for centered, x=B for make left-justified block, then center whole block. Implies .KS.
.EG	no	-	Print document in BTL format for 'Engineer's Notes.' Must be first.
.EN	-	yes	Space after equation produced by eqn or neqn.
.EQ x y	-	yes	Precede equation; break out and add space. Equation number is y. The optional argument x may be I to indent equation (default), L to left-adjust the equation, or C to center the equation.
.FE	-	yes	End footnote.
.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.HO	-	no	'Bell Laboratories, Holmdel, New Jersey 07733'.
.I x	no	no	Italicize x; if x missing, italic text follows.
.IH	no	no	'Bell Laboratories, Naperville, Illinois 60540'
.IM	no	no	Print document in BTL format for an internal memorandum. Must be first.
.IP x y	no	yes	Start indented paragraph, with hanging tag x. Indentation is y ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MF	-	-	Print document in BTL format for 'Memorandum for File.' Must be first.
.MH	-	no	'Bell Laboratories, Murray Hill, New Jersey 07974'.
.MR	-	-	Print document in BTL format for 'Memorandum for Record.' Must be first.
.ND date	troff	no	Use date supplied (if any) only in special BTL format positions; omit from page footer.
.NH n	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where n tells what level is wanted (default is 1).

.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.PT	pg #	-	Page title, automatically invoked at top of page. May be redefined.
.PY	-	no	'Bell Laboratories, Piscataway, New Jersey 08854'
.QE	-	yes	End quoted (indented and shorter) material.
.QP	-	yes	Begin single paragraph which is indented and shorter.
.QS	-	yes	Begin quoted (indented and shorter) material.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's are measured from current indentation.
.SG x	no	yes	Insert signature(s) of author(s), ignored except in TM. x is the reference line (initials of author and typist).
.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TA x...	5...	no	Set tabs in ens. Default is 5 10 15 ...
.TE	-	yes	End table.
.TH	-	yes	End heading section of table.
.TL	no	yes	Title follows.
.TM x...	no	-	Print document in BTL technical memorandum format. Arguments are TM number, (quoted list of) case number(s), and file number. Must precede other requests.
.TR x	-	-	Print in BTL technical report format; report number is x. Must be first.
.TS x	-	yes	Begin table; if x is H table has repeated heading.
.UL x	-	no	Underline argument (even in troff).
.UX	-	no	'UNIX'; first time used, add footnote 'UNIX is a trademark of Bell Laboratories.'
.WH	-	no	'Bell Laboratories, Whippany, New Jersey 07981'.

NAME

terminals - conventional names

DESCRIPTION

These names are used by certain commands and are maintained as part of the shell environment (see _sh_(1),_environ_(5)).

adm3a	Lear Seigler Adm-3a
2621	Hewlett-Packard HP262? series terminals
hp	Hewlett-Packard HP264? series terminals
c100	Human Designed Systems Concept 100
h19	Heathkit H19
mime	Microterm mime in enhanced ACT IV mode
1620	DIABLO 1620 (and others using HyType II)
300	DASI/DTC/GSI 300 (and others using HyType I)
33	TELETYPE(Reg.) Model 33
37	TELETYPE Model 37
43	TELETYPE Model 43
735	Texas Instruments TI735 (and TI725)
745	Texas Instruments TI745
dumb	terminals with no special features
4014	Tektronix 4014
vt52	Digital Equipment Corp. VT52

The list goes on and on. Consult /etc/termcap (see _termcap_(5)) for an up-to-date and locally correct list.

Commands whose behavior may depend on the terminal either consult TERM in the environment, or accept arguments of the form -Tterm, where _term_ is one of the names given above.

SEE ALSO

stty(1), tabs(1), plot(1), sh(1), environ(5) ex(1), clear(1), more(1), ul(1), tset(1), termcap(5), ttytype(5)
troff(1) for _nroff_

BUGS

The programs that ought to adhere to this nomenclature do so only fitfully.

NAME

boot - startup procedures

DESCRIPTION

A 68000 UNIX system is typically started by a two-stage process. The first is a primary bootstrap which is used to read in the system itself.

The primary bootstrap, when read into memory and executed, sets up memory management if necessary, and types a prompt message on the console. Then it reads from the console a device specification (see below) followed immediately by a pathname. This program finds the corresponding file on the given device, loads that file into the proper memory location, and then transfers control of the program. Normal line editing characters can be used.

Conventionally, the name of the current version of the system is '/unix'. Then, the recipe is:

1) Load the boot program by fiddling with the console keys and crt as appropriate for your hardware.

2) When the prompt is given, type [for example]
fpy(0,0)unix
or
hd(0,0)unix
depending on whether you are loading from floppy or hard disk, respectively. The first 0 indicates the physical unit number; the second indicates the block number of the beginning of the logical file system (device) to be searched. (See below).

When the system is running, it types a '#' prompt. After doing any file system checks via fsck(1) and setting the date (date(1)), the system can be brought up for standard operation by typing an EOT (control-d) in response to the '#' prompt.

Device specifications .

A device specification has the following form:

```
device(unit,offset)
```

where device is the type of the device to be searched, unit is the unit number of the device, and offset is the block offset of the file system on the device. Device specifications vary according to which 68000 UNIX system you are using. Check manufacturers' instructions for the device specifications.

For example, the specification

```
hp(1,7000)
```

would indicate an RP03 disk, unit 1, and the file system found starting at block 7000 (cylinder 35).

ROM Programs .

Programs to call the primary bootstrap may be installed in read-only memories or manually keyed into main memory. Each program is position-independent but should be placed well above location 0 so it will not be overwritten. See manufacturer's instructions for a manually keyed-in ROM boot program, should one become necessary.

FILES

/unix - system code

SEE ALSO

init(1M)

NAME

ident - login banner

SYNOPSIS

/etc/ident

DESCRIPTION

/*etc*/*ident* contains the login banner for the 68000 system that gets printed on the user's terminal before a user enters his/her login name. /*etc*/*ident* usually includes the company name and other pertinent information.

NAME

rc - command script for system housekeeping

SYNOPSIS

/etc/rc

DESCRIPTION

The /*etc*/*rc* program is called immediately after the system is booted. Its responsibility is to clear the records of what devices and what users were present on the system when it was last running.

These housekeeping functions include mounting default devices and calling /*etc*/*update*, *cron*, and user accounting programs.

SEE ALSO

init(1M)