VA-003, Rev 2

UNIX PROGRAMMER'S MANUAL

Seventh Edition
Virtual VAX−11 Version

June, 1981

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

This manual reflects system enhancements made at Berkeley and sponsored in part by NSF Grants MCS-7807291, MCS-8005144, and MCS-74-07644-A03; DOE Contract DE-AT03-76SF00034 and Project Agreement DE-AS03-79ER10358; and by Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, Monitored by Naval Electronics Systems Command Under Contract No. N00039-80-K-0649.

PREFACE

This update to the fourth distribution of November 1980 provides support for the VAX 11/750 and for the full interconnect architecture of the VAX 11/780. Robert Elz of the University of Melbourne contributed greatly to this distribution especially in the boot-time system configuration code; Bill Shannon of DEC supplied us with the implementation of DEC standard bad block handing. The research group at Bell Laboratories and DEC Merrimack provided us with access to 11/750's in order to debug its support.

Other individuals too numerous to mention provided us with bug reports, fixes and other enhancements which are reflected in the system. We are ever grateful to the UNIX user community for encouragement and support.

The financial support of the Defense Advanced Research Projects Agency in support of this work is gratefully acknowledged.

W. N. Joy
R. S. Fabry
K. Sklower

Preface to the Fourth Berkeley distribution

This manual reflects the Berkeley system mid-October, 1980. A large amount of tuning has been done in the system since the last release; we hope this provides as noticeable an improvement for you as it did for us. This release finds the system in transition; a number of facilities have been added in experimental versions (job control, resource limits) and the implementation of others is imminent (shared-segments, higher performance from the file system, etc.). Applications which use facilities that are in transition should be aware that some of the system calls and library routines will change in the near future. We have tried to be conscientious and make it very clear where this is likely.

A new group has been formed at Berkeley, to assume responsibility for the future development and support of a version of UNIX on the VAX. The group has received funding from the Defense Advanced Research Projects Agency (DARPA) to supply a standard version of the system to DARPA contractors. The same version of the system will be made available to other licensees of UNIX on the VAX for a duplication charge. We gratefully acknowledge the support of this contract.

We wish to acknowledge the contribution of a number of individuals to the the system.

We would especially like to thank Jim Kulp of IIASA, Laxenburg Austria and his colleagues, who first put job control facilities into UNIX; Eric Allman, Robert Henry, Peter Kessler and Kirk McKusick, who contributed major new pieces of software; Mark Horton, who contributed to the improvement of facilities and substantially improved the quality of our bit-mapped fonts, our hardware support staff: Bob Kridle, Anita Hirsch, Len Edmondson and Fred Archibald, who helped us to debug a number of new peripherals; Ken Arnold who did much of the leg-work in getting this version of the manual prepared, and did the final editing of sections 2-6, some special individuals within Bell Laboratories: Greg Chesson, Stuart Feldman, Dick Haight, Howard Katseff, Brian Kernighan, Tom London, John Reiser, Dennis Ritchie, Ken Thompson, and Peter Weinberger who helped out by answering questions; our excellent local DEC field service people, Kevin Althaus and Frank Chargois who kept our machine running virtually all the time, and fixed it quickly when things broke; and, Mike Accetta of Carnegie-Mellon University, Robert Elz of the University of Melbourne, George Goble of Purdue University, and David Kashtan of the Stanford Research Institute for their technical advice and support.

Special thanks to Bill Munson of DEC who helped by augmenting our computing facility and to Eric Allman for carefully proofreading the "last" draft of the manual and finding the bugs which we knew were there but couldn't see.

We dedicate this to the memory of David Sakrison, late chairman of our department, who gave his support to the establishment of our VAX computing facility, and to our department as a whole.

W. N. Joy
O. Babaoğlu
R. S. Fabry
K. Sklower

Preface to the Third Berkeley distribution

This manual reflects the state of the Berkeley system, December 1979. We would like to thank all the people at Berkeley who have contributed to the system, and particularly thank Prof. Richard Fateman for creating and administrating a hospitable environment, Mark Horton who helped prepare this manual, and Eric Allman, Bob Kridle, Juan Porcar and Richard Tuck for their contributions to the kernel.

The cooperation of Bell Laboratories in providing us with an early version of UNIX/32V is greatly appreciated. We would especially like to thank Dr. Charles Roberts of Bell Laboratories for helping us obtain this release, and acknowledge T. B. London, J. F. Reiser, K. Thompson, D. M. Ritchie, G. Chesson and H. P. Katseff for their advice and support.

W. N. Joy
O. Babaoğlu

Preface to the UNIX/32V distribution

The UNIX† operating system for the VAX*-11 provides substantially the same facilities as the UNIX system for the PDP*-11.

We acknowledge the work of many who came before us, and particularly thank G. K. Swanson, W. M. Cardoza, D. K. Sharma, and J. F. Jarvis for assistance with the implementation for the VAX-11/780.

T. B. London
J. F. Reiser

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

†UNIX is a Trademark of Bell Laboratories.
*VAX and PDP are Trademarks of Digital Equipment Corporation.

INTRODUCTION TO VOLUME 1

This volume gives descriptions of the publicly available features of the UNIX/32V† system, as extended to provide a virtual memory environment and other enhancements at U. C. Berkeley. It does not attempt to provide perspective or tutorial information upon the UNIX operating system, its facilities, or its implementation. Various documents on those topics are contained in Volume 2. In particular, for an overview see 'The UNIX Time-Sharing System' by Ritchie and Thompson; for a tutorial see 'UNIX for Beginners' by Kernighan, and for an guide to the new features of this virtual version, see 'Getting started with Berkeley Software for UNIX on the VAX' in volume 2c.

Within the area it surveys, this volume attempts to be timely, complete and concise. Where the latter two objectives conflict, the obvious is often left unsaid in favor of brevity. It is intended that each program be described as it is, not as it should be. Inevitably, this means that various sections will soon be out of date.

The volume 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 contradistinction to subroutines, which are intended to be called by the user's programs. Commands generally reside in directory */bin* (for *bin*ary programs). Some programs also reside in */usr/bin,* or in */usr/ucb,* to save space in */bin.* These directories are searched automatically by the command interpreters.

System calls are entries into the UNIX supervisor. The system call interface is identical to a C language procedure call; the equivalent C procedures are described 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). The functions are described in terms of C, but most will work with Fortran as well.

The special files section 4 discusses the characteristics of each system 'file' that actually refers to an I/O device. The names in this section refer to the DEC device names for the hardware, instead of the names of the special files themselves.

The file formats and conventions section 5 documents the structure of particular kinds of files; for example, the form of the output of the loader and assembler is given. Excluded are files used by only one command, for example the assembler's intermediate files.

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

†UNIX is a Trademark of Bell Laboratories.

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

The maintenance section 8 discusses commands and procedures not intended for use by the ordinary user. The commands and files described here are almost all kept in the directory */etc.*

Each section consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, together with the section number, and sometimes a letter characteristic of a subcategory, e.g. graphics is 1G, and the math library is 3M. Entries within each section are alphabetized. The page numbers of each entry start at 1; it is infeasible to number consecutively the pages of a document like this that is republished in many variant forms.

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 subsection:
>
>> **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 *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.
>
> In section 2 an *assembler* subsection carries the PDP-11 assembly-language system interface.

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.

Logging in. You must call UNIX from an appropriate terminal. Most any ASCII terminal capable of full duplex operation and generating the entire character set can be used. You must also have a valid user name, which may be obtained, together with necessary telephone numbers, from the system administration. After a data connection is established, the login procedure depends on what kind of terminal you are using.

300-baud terminals: Such terminals include the GE Terminet 300, and most display terminals run with popular modems. These terminals generally have a speed switch which should be set at '300' (or '30' for 30 characters per second) and a half/full duplex switch which should be set at full-duplex. (This switch will often have to be changed since many other systems require half-duplex). When a connection is established, the system types 'login:'; you type your user name, followed by the 'return' key. If you have a password, the system asks for it and turns off the printer on the terminal so the password will not appear. After you have logged in, the 'return', 'new line', or 'linefeed' keys will give exactly the same results.

1200- and 150-baud terminals: If there is a half/full duplex switch, set it at full-duplex. When you have established a data connection, the system types out a few garbage characters (the 'login:' message at the wrong speed). Depress the 'break' (or 'interrupt') key; this is a speed-independent signal to UNIX that a different speed terminal is in use. The system then will type 'login:,' this time at another speed. Continue depressing the break key until 'login:' appears in clear, then respond with your user name. From the TTY 37 terminal, and any other which has the 'newline' function (combined carriage return and linefeed), terminate each line you type with the 'new line' key, otherwise use the 'return' key.

Hard-wired terminals. Hard-wired terminals usually begin at the right speed, up to 9600 baud; otherwise the preceding instructions apply.

For all these terminals, it is important that you type your name in lower-case if possible; if you type upper-case letters, UNIX will assume that your terminal cannot generate lower-case letters and will translate all subsequent upper-case letters to lower case.

The evidence that you have successfully logged in is that a shell program will type a prompt ('$' or '%') to you. (The shells are described below under 'How to run a program.')

For more information, consult *tset*(1), and *stty*(1), which tell how to adjust terminal behavior, *getty*(8), which discusses the login sequence in more detail, and *tty*(4), which discusses terminal I/O.

Logging out. There are three 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.
>
> You can log in directly as another user by giving a *login*(1) command.
>
> If worse comes to worse, you can simply hang up the phone; but beware — some machines may lack the necessary hardware to detect that the phone has been hung up. Ask your system administrator if this is a problem on your machine.

How to communicate through your terminal. When you type characters, a gnome deep in the system gathers your characters and saves them in a secret place. The characters will not be given to a program until you type a return (or newline), as described above in *Logging in.*

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. In a newer terminal driver, it is possible to cause output to be thrown away without interrupting the program by typing ^O; see *newtty*(4).

The *quit* 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 but also generates a file with the core image of the terminated process. Quit is useful for debugging.

Besides adapting to the speed of the terminal, UNIX tries to be intelligent about whether you have a terminal with the 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, the *reset*(1) command will rescue you.

Tab characters are used freely in UNIX source programs. If your terminal does not have the tab function, you can arrange to have 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.

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). If the shell prompts you with '$', then it is an instance of *sh*(1) the standard Bell-labs provided shell. If it prompts with '%' then it is an instance of *csh*(1) a shell written at Berkeley. The shells are different for all but the most simple terminal usage. Most users at Berkeley choose *csh*(1) because of the *history* mechanism and the *alias* feature, which greatly enhance its power when used interactively. *Csh* also supports the job-control facilities new to this release of the system. See *newcsh*(1) or the Csh introduction in volume 2C for details.

You can change from one shell to the other by using the *chsh*(1) command, which takes effect at your next login.

The current directory. UNIX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he also created a directory for you (ordinarily with the same name as your user name). When you log in, any file name you type is by default in this directory. Since you are the owner of this directory, you have full 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.

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, */usr/lem/filex* refers to the file *filex* in the directory *lem; lem* is itself a subdirectory of *usr; usr* springs directly from the root directory.

If your current directory has subdirectories, the path names of files therein begin with the name of the 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.

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 *f77*(1), the Pascal compiler *pc*(1), and interpreter *pi*(1) and *px*(1), the Lisp system *lisp*(1), and the APL system *apl*(1). After the program text has been entered through the editor and written on a file, you can give the file to the appropriate language processor as an argument. The output of the language processor will be left on a file in the current directory named 'a.out'. (If the output is precious, use *mv* to move it to a less exposed name soon.)

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 ('$' or '%') 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 write text on a terminal 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 multi-column output. *Nroff* is an elaborate text formatting program. Used naked, it requires careful forethought, but for ordinary documents it has been tamed; see *me*(7) and *ms*(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. It was used to produce this manual.

Script(1) lets you keep a record of your session in a file, which can then be printed, mailed, etc. It provides the advantages of a hard-copy terminal even when using a display terminal.

More(1) is useful for preventing the output of a command from zipping off the top of your screen. It is also well suited to perusing files.

Status inquiries. Various commands exist to provide you with useful information. *w*(1) prints a list of users presently logged in, and what they are doing. *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.

If you use *csh*(1) the key ^Z (control-Z) will cause jobs to "stop". If this happens before you learn about it, you can simply continue by saying "fg" (for foreground) to bring the job back.

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

CONVERTING FROM THE 6TH EDITION

There follows a catalogue of significant, mostly incompatible, changes that will affect old users converting from the sixth edition on a PDP-11. No attempt is made to list all new facilities, or even all minor, but easily spotted changes, just the bare essentials without which it will be almost impossible to do anything.

Addressing files. Byte addresses in files are now long (32-bit) integers. Accordingly *seek* has been replaced by *lseek*(2). Every program that contains a *seek* must be modified. *Stat* and *fstat*(2) have been affected similarly, since file lengths are now 32- rather than 24-bit quantities.

Assembly language. This language is dead. Necromancy will be severely punnished.

Stty and *gtty.* These system calls have been extensively altered, see *ioctl*(2) and *tty*(4).

C language, lint. The syntax for initialization requires an equal sign = before an initializer, and brackets { } around compound initial values; arrays and structures are now initialized honestly. Assignment operators such as =+ and =− are now written in the reverse order: +=, −=. This removes the possibility of ambiguity in constructs such as x=−2, y=*p, and a=/*b. You will also certainly want to learn about

long integers
type definitions
casts (for type conversion)
unions (for more honest storage sharing)
#include <filename> (which searches in standard places)

The program *lint*(1) checks for obsolete syntax and does strong type checking of C programs, singly or in groups that are expected to be loaded together. It is indispensable for conversion work.

Fortran. The old *fc* is replaced by *f77*, a true compiler for Fortran 77, compatible with C. There are substantial changes in the language; see 'A Portable Fortran 77 Compiler' in Volume 2.

Stream editor. The program *sed*(1) is adapted to massive, repetitive editing jobs of the sort encountered in converting to the new system. It is well worth learning.

Standard I/O. The old *fopen, getc, putc* complex and the old *−lp* package are both dead, and even *getchar* has changed. All have been replaced by the clean, highly efficient, *stdio*(3) package. The first things to know are that *getchar*(3) returns the integer EOF (−1) (which is not a possible byte value) on end of file, that 518-byte buffers are out, and that there is a defined FILE data type.

Make. The program *make*(1) handles the recompilation and loading of software in an orderly way from a 'makefile' recipe given for each piece of software. It remakes only as much as the modification dates of the input files show is necessary. The makefiles will guide you in building your new system.

Shell, chdir. F. L. Bauer once said Algol 68 is the Everest that must be climbed by every computer scientist because it is there. So it is with the shell for UNIX users. Everything beyond simple command invocation from a terminal is different. Even *chdir* is now spelled *cd.* If you wish to use *sh* (as opposed to *csh*) then you will want to study *sh*(1) long and hard.

C shell. *Csh*(1), developed at Berkeley, has features comparible to *sh*. It includes a history mechanism that saves you from retyping all or part of previous commands, as well as an efficient aliasing (macro) mechanism. The job control facilities of the system, which make the system much more pleasant to use, are currently available only with *csh*. See *newcsh*(1) for a description. These features make *csh* pleasant to use interactively. *Csh* programs have a syntax reminiscent of *C*, while *sh* command programs have a syntax reminiscent of ALGOL-68.

Debugging. *Sdb*(1) is a far more capable replacement for the debugger *cdb*, and debugs C and Fortran at the source level. For machine language debugging, *adb* replaces *db*. The first-time user should be especially careful about distinguishing / and ? in *adb* commands, and watching to make sure that the *x* whose value he asked for is the real *x*, and not just some absolute location equal to the stack offset of some automatic *x*. You can always use the 'true' name, *_x*, to pin down a C external variable.

Dsw. This little-known, but indispensable facility has been taken over by *rm −ri*.

Boot procedures. Needless to say, these are all different. See section 8 of this volume, and the other documentation you should have received with your tape.

CONVERTING FROM THE DECEMBER, 1979 BERKELEY DISTRIBUTION

There have been a number of significant changes and improvements in the system. This list just gives the bare essentials:

C language changes. The C compiler now accepts and checks essentially arbitrary length identifiers and preprocessor names. There is a new type available in type casts: **void** which signifies that a value is to be ignored. It is useful in keeping lint happy about values which are not used (especially values returned from procedures). Finally, the language has been changed so that field names need not be unique to structures; on the other hand, the compiler insists that you be more honest about types involved in pointer constructs or it will warn you.

Object file format. The object file format has been changed to include a string table, so that language compilers may have names longer than 8 characters in their resulting *a.out* files. Old *.o* files must be recreated. *A.out* files will still run on both this and the December 1979 version of the system; only the symbol tables are incompatible.

Archive format and table of contents. The archive format has been changed to one which is portable between the VAX and other machines (e.g. the PDP-11). Old VAX archives should be converted with *arcv*(8); loader archives should just be recreated since the object files are also obsolete. Loader archives should have table-of-contents added by *ranlib*(1); if they dont the loader will gripe when they are used. See also *old*(8).

New tty driver, job control facilities and csh. Hand in hand are new job control facilities, a new tty driver and a new version of the C shell which supports and uses all of this. See *newtty*(4) and *newcsh*(1) for a quick introduction. You should use *oldcsh* until you learn about the new facilities.

Pascal compiler. There is a true Pascal compiler, *pc*(1) which allows separate compilation as well as mixing in of FORTRAN and C code.

Error analyzer. There is an error analyzer program *error*(1), which takes a set of error message and merges them back into the source files at the point of error. It can be used interactively to avoid inserting errors which are uninteresting. This program eliminates once and for all making lists of errors on small scraps of paper.

Mail forwarding. The system now provides mail forwarding and distribution facilities. Group and aliases are defined in the file */usr/lib/aliases* see *aliases*(5). If you change this file you will have to rerun *newaliases*(1). For any particular system a table in the source of the *delivermail* postman program may have to be changed so that it knows about the gateways on the local machine.

System bootstrap procedures. These are totally changed; the system performs automatic reboots and preens the disks automatically at reboot. You should reread the appropriate pages in section 8 if you deal with system reboots.

TABLE OF CONTENTS

1. Commands and Application Programs

2. System Calls

3. Subroutines

4. Special Files

5. File Formats

6. Games

7. Miscellaneous

8. System Maintenance

PERMUTED INDEX

NAME

intro — introduction to commands

DESCRIPTION

This section describes publicly accessible commands in alphabetic order. Certain distinctions of purpose are made in the headings:

(1) Commands of general utility.

(1C) Commands for communication with other systems.

(1G) Commands used primarily for graphics and computer-aided design.

N.B.: Commands related to system maintenance, which appeared in section 1, distinguished by (1M), in previous versions of the manual have been moved to section 8, as they are of little interest to most users.

The word 'VAX-11' at the foot of a page means that some or all of the description applies only to the implementation for the Digital Equipment Corporation VAX-11. Pages added or changed between the distribution of UNIX/32V and the Berkeley Distribution indicate '3rd Berkeley Distribution' or '4th Berkeley Distribution' at the lower left, as appropriate.

SEE ALSO

Section (6) for computer games, section (8) for system maintenance commands.

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* 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.

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 which start at 0x80000000. 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 prefixes 0o and 0O ("zero oh") force interpretation in octal radix; the prefixes 0t and 0T force interpretation in decimal radix; the prefixes 0x and 0X force interpretation in hexadecimal radix. Thus 0o20 = 0t16 = 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 disinguish 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.* (This form is currently broken on the VAX; local variables can be examined only with *sdb*(1).)

(*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.

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)).
i	n	Print as VAX instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a	0	Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below.

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

p	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 r5). If C is used then the names and (16 bit) values of all automatic and static variables are printed for each active function. If *count* is given then only the first *count* frames are printed.

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.

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

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

k
: The current subprocess, if any, is terminated.

VARIABLES

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

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.
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 or 0413).
s The stack segment size.
t The text segment size.

ADDRESSES

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

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

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

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

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

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

FILES

a.out
core

SEE ALSO

sdb(1), ptrace(2), a.out(5), core(5)

DIAGNOSTICS

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

BUGS

Local variable addresses and names are recorded in the *a.out* file in a format known only to *sdb*(1).

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

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

NAME
apl − an apl interpreter

SYNOPSIS
apl

DESCRIPTION
Apl is an APL interpreter. All of the operators are exactly as in apl\360. Overstrikes are often required, and they work (use ctrl-h).

Function definition is not what you would expect. Functions are loaded from files. The first line of the file is the function header, as you would expect it but with no del. The rest of the file is the lines of the function. Lines are numbered, but there are no square brackets with line numbers. If you say)READ FILE it will load the function in that file. If you say)EX FILE it will put you in the editor to change that file. Upon exit, it will read the file in as though by)READ.

All of the usual operators are available, including domino. Also available are monadic encode and epsilon.

The following *apl* system commands are available.

)ASCII
: changes terminal to accept and print ASCII characters and operators; this is the default. If you are stuck in APL mode on an ASCII terminal, '"' is ')' and lowercase letters map to uppercase.

)APL
: changes terminal to accept and print APL characters. Erase is set to Ω and kill is set to α.

)DIGITS n
: sets the number of digits displayed to n, from 1 to 19.

)FUZZ n
: sets the fuzz to n.

)ORIGIN n
: sets the origin to n, which should be 1 or 0.

)WIDTH n
: sets *apl*'s idea of your terminal's carriage width.

)ERASE n
: gets rid of function or variable named n.

)SAVE n
: saves all variables and functions (workspace) in file named n. Workspaces are sensitive to changes in *apl*.

)LOAD n
: gets the workspace in file n (which must have been)SAVE'd.)

)COPY n
: like)LOAD but variables and functions are not erased. Things in the loaded file take precedence over stuff already in.

)CLEAR
: clears the workspace.

)DROP n
: deletes file n in your directory, which need not be saved from *apl*.

)CONTINUE

exits and saves workspace in file *continue* which is loaded next time you run *apl.*

)OFF
: exits *apl.*

)READ n
: reads in a function from file *n.* The first line is the header, with no del's. The full APL360 header is accepted. All other lines in the file are lines in the function. Lines are implicitly numbered, and transfers are as usual. There are no labels.

)EDIT n
: runs the editor *ed*(1) on file *n*, and then)READ's the file when you leave the editor.

)EX n
: runs the editor *ex*(1) on file *n*, and then)READ's the file when you leave the editor.

)VI n
: runs the editor *vi*(1) on file *n*, and then)READ's the file when you leave the editor.

)LIB
: lists out all of the files in the current directory.

)FNS
: lists out all current functions.

)VARS
: lists out all current variables.

)DEBUG
: toggles a debugging switch, which can produce vast amounts of hopelessly cryptic output.

FILES

apl_ws — temporary workspace file
continue — continue workspace

AUTHORS

Ken Thompson, Ross Harvey, Douglas Lanam

BUGS

This program has not been extensively used or tested.

ASCII CHAR MNEMONICS

&	∧	and	#	×	times
−	−	minus	+	+	add
<	<	less than	>	>	greater than
=	=	equal to	,	,	comma
%	÷	divide	*	*	exponential (power)
!	!	factorial and combinations	?	?	deal
.le	≤	less than or equal	.ge	≥	greater than or equal
.ne	≠	not equal	.om	Ω	omega (not used)
.ep	∊	epsilon	.rh	⍴	shape (rho)
.nt	¬	not (also '~')	.tk	↑	take (also '^')
.dr	↓	drop	.it	⍳	iota
.ci	○	circular function	.al	⍺	alpha (not used)
.cl	⌈	maximum (ceiling)	.fl	⌊	minimum (floor)
.dl	∆	del (not used)	.de	∇	upside down del
.jt	∘	small circle (null)	.qd	⎕	quad
.ss	⊂	right U (not used)	.sc	⊃	left U (not used)
.si	∩	Down U	.su	∪	U (not used)
.[ˆ	∇	upside-down del	.bv	⊥	decode (base)
.rp	⊤	encode (rep)	.br	∣	residue (mod)
.sp	←	assignment (also '_')	.go	→	goto
.or	∨	or	.nn	⍲	nand
.nr	⍱	nor	.lg	⍟	log
.rv	⌽	reversal	.tr	⍉	transpose
.rb		reverse bar	.cb	⍪	comma bar (not used)
.sb	⌿	slash bar	.bb	⍀	blackslash bar
.gu	⍋	grade up	.gd	⍒	grade down
.qq	⍞	quote quad	.dm	⌹	domino
.lm	⍝	lamp	.ib	⌶	I-beam
.ex		execute (not used)	.fr		format(not used)
.di		diamond (not used)	.ot		out (not used)
.ld	⍫	locked del (not used)	._a	A	alias for 'A'
._b	B	alias for 'B'	._c	C	alias for 'C'
._d	D	alias for 'D'	._e	E	alias for 'E'
._f	F	alias for 'F'	._g	G	alias for 'G'
._h	H	alias for 'H'	._i	I	alias for 'I'
._j	J	alias for 'J'	._k	K	alias for 'K'
._l	L	alias for 'L'	._m	M	alias for 'M'
._n	N	alias for 'N'	._o	O	alias for 'O'
._p	P	alias for 'P'	._q	Q	alias for 'Q'
._r	R	alias for 'R'	._s	S	alias for 'S'
._t	T	alias for 'T'	._u	U	alias for 'U'
._v	V	alias for 'V'	._w	W	alias for 'W'
._x	X	alias for 'X'	._y	Y	alias for 'Y'
._z	Z	alias for 'Z'			

NAME

apropos — locate commands by keyword lookup

SYNOPSIS

apropos keyword ...

DESCRIPTION

Apropos shows which manual sections contain instances of any of the given keywords in their title. Each word is considered separately and case of letters is ignored. Words which are part of other words are considered thus looking for compile will hit all instances of 'compiler' also. Try

    apropos password

and

    apropos editor

If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. Try 'apropos format' and then 'man 3s printf' to get the manual on the subroutine *printf.*

Apropos is actually just the **−k** option to the *man*(2) command.

FILES

/usr/lib/whatis data base

SEE ALSO

man(1), whatis(1), catman(8)

AUTHOR

William Joy

NAME

ar — archive and library maintainer

SYNOPSIS

ar key [posname] afile name ...

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose. **N.B:** This version of *ar* uses a ASCII-format archive which is portable among the various machines running UNIX. Programs for dealing with older formats are available: see *old*(8).

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

d Delete the named files from the archive file.

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

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

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

p Print the named files in the archive.

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

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

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

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

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

FILES

/tmp/v* temporaries

SEE ALSO

lorder(1), ld(1), ranlib(1), ar(5), arcv(8), old(8)

BUGS

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

NAME

as − assembler

SYNOPSIS

as [**−d124**] [**−L**] [**−W**] [**−V**] [**−J**] [**−R**] [**−t** directory] [**−o** objfile] [name ...]

DESCRIPTION

As assembles the named files, or the standard input if no file name is specified. The available flags are:

−d Specifies the number of bytes to be assembled for offsets which involve forward or external references, and which have sizes unspecified in the assembly language. The default is **−d4.**

−L Save defined labels beginning with a 'L', which are normally discarded to save space in the resultant symbol table. The compilers generate such temporary labels.

−V Use virtual memory for intermediate storage, rather than a temporary file.

−W Do not complain about errors.

−J Use long branches to resolve jumps when byte-displacement branches are insufficient. This must be used when a compiler-generated assembly contains branches of more than 32k bytes.

−R Make initialized data segments read-only, by concatenating them to the text segments. This obviates the need to run editor scripts on assembly code to make initialized data read-only and shared.

−t Specifies a directory to receive the temporary file, other than the default /tmp.

All undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file *objfile;* if that is omitted, *a.out* is used.

FILES

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

SEE ALSO

ld(1), nm(1), adb(1), sdb(1), a.out(5)
Auxiliary documentation *Assembler Reference Manual.*

AUTHORS

John F. Reiser
Robert R. Henry

BUGS

−J should be eliminated; the assembler should automatically choose among byte, word and long branches.

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) (or *csh*(1) if you normally use it) at a specified later time. A *cd* command to the current directory is inserted at the beginning, followed by assignments to all environment variables (excepting the variable TERMCAP, which is useless in this context.) When the script is run, it uses the user and group ID of the creator of the copy file.

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

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

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

At programs are executed by periodic execution of the command */usr/lib/atrun* from *cron*(8). The granularity of *at* depends upon how often *atrun* is executed.

Standard output or error output is lost unless redirected.

FILES

/usr/lib/atrun	executor (run by *cron*(8)).
in /usr/spool/at:	
yy.ddd.hhhh.*	activity for year yy, day dd, hour hhhh.
lasttimedone	last *hhhh*
past	activities in progress

SEE ALSO
calendar(1), pwd(1), sleep(1), cron(8)

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] [prog] [file] ...

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as **−f** *file*.

Files are read in order; if there are no files, the standard input is read. The file name '−' means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, *vide infra*.) The fields are denoted $1, $2, ... ; $0 refers to the entire line.

A pattern-action statement has the form

```
pattern { action }
```

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

An action is a sequence of statements. A statement can be one of the following:

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

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

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

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer. *substr(s, m, n)* returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf(fmt, expr, expr, ...)* formats the expressions according to the *printf*(3) format given by *fmt* and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in *egrep*. Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

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

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

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

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

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

or by using the **−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").

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

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

Print fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

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

SEE ALSO

lex(1), sed(1)

A. V. Aho, B. W. Kernighan, P. J. Weinberger, *Awk — a pattern scanning and processing language*

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

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.

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

Comments
    are enclosed in /* and */.

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

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

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

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

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

Functions in −l math library
```
s(x)    sine
c(x)    cosine
e(x)    exponential
l(x)    log
a(x)    arctangent
j(n,x)  Bessel function
```

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or newlines may separate statements. Assignment to *scale* influences the number of digits to be retained on arithmetic operations in the manner of *dc*(1). Assignments to *ibase* or *obase* set the input and output number radix respectively.

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

For example

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

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

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

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

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

FILES

/usr/lib/lib.b mathematical library
dc(1) desk calculator proper

SEE ALSO

dc(1)
L. L. Cherry and R. Morris, *BC − An arbitrary precision desk-calculator language*

BUGS

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

NAME

biff — be notified if mail arrives and who it is from

SYNOPSIS

biff [**yn**]

DESCRIPTION

Biff informs the system whether you want to be notified when mail arrives during the current terminal session. The command

biff y

enables notification; the command

biff n

disables it. When mail notification is enabled, the header and first few lines of the message will be printed on your screen whenever mail arrives. A "biff y" command is often included in the file *.login* or *.profile* to be executed at each login.

Biff operates asynchronously. For synchronous notification use the MAIL variable of *sh*(1) or the *mail* variable of *csh*(1).

SEE ALSO

csh(1), sh(1), mail(1), mail (4)

BUGS

NAME

mail − send or receive mail among users

SYNOPSIS

/bin/mail [+] [−**i**] [person] ...
/bin/mail [+] [−**i**] −**f** file

DESCRIPTION

Note: This is one of two mailers with the name *mail.* The default *mail* command is described in *mail*(1), and its binary is in the directory */usr/ucb.*

Mail with no argument prints a user's mail, message-by-message, in last-in, first-out order; the optional argument + causes first-in, first-out order. For each message, it 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.

!*command*
: Escape to the Shell to do *command.*

*
: Print a command summary.

An interrupt normally causes termination of the command; the mail file is unchanged. The optional argument −i causes *mail* to continue after interrupts.

When *persons* are named, *mail* takes the standard input up to an end-of-file (or a line with just '.') and adds it to each *person's* 'mail' file. The message is preceded by the sender's name and a postmark. Lines that look like postmarks are prepended with '>'. A *person* is usually a user name recognized by *login*(1). To denote a recipient on a remote system, prefix *person* by the system name and exclamation mark (see *uucp*(1)).

The −**f** option causes the named file, e.g. 'mbox', to be printed as if it were the mail file.

When a user logs in he is informed of the presence of mail.

FILES

/etc/passwd	to identify sender and locate persons
/usr/spool/mail/*	incoming mail for user *
mbox	saved mail
/tmp/ma*	temp file
/usr/spool/mail/*.lock	lock for mail directory
dead.letter	unmailable text

SEE ALSO

mail(1), write(1), uucp(1), uux(1), xsend(1), delivermail(8)

BUGS

Race conditions sometimes result in a failure to remove a lock file.

Normally anybody can read your mail, unless it is sent by *xsend*(1). An installation can overcome this by making *mail* a set-user-id command that owns the mail directory.

NAME

cal − print calendar

SYNOPSIS

cal [month] year

DESCRIPTION

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

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.
Beware that 'cal 78' 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*(8).

FILES

calendar
/usr/lib/calendar to figure out today's and tomorrow's dates
/etc/passwd
/tmp/cal*
egrep, sed, mail subprocesses

SEE ALSO

at(1), cron(8), mail(1)

BUGS

Your calendar must be public information for you to get reminder service.
Calendar's extended idea of 'tomorrow' doesn't account for holidays.

NAME

call — ring a telephone

SYNOPSIS

call phonenumber

DESCRIPTION

Call places an outgoing call to the specified *phonenumber*. Nothing special happens when the called party answers. *Phonenumber* may have any number of digits; a '+' sign may be used to specify a point at which to wait for a second dial tone.

FILES

/dev/dn0

SEE ALSO

cu(1), dn(5)

NAME

cat − catenate and print

SYNOPSIS

cat [**−u**] [**−n**] [**−s**] [**−v**] 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 1024-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.

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.

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

DESCRIPTION

Cb places a copy of the C program from the standard input on the standard output with spacing and indentation that displays the structure of the program.

BUGS

NAME

cc − C compiler

SYNOPSIS

cc [option] ... file ...

DESCRIPTION

Cc is the UNIX C compiler. **N.B.:** *cc* has been recently modified to accept arbitrary length identifiers and to produce a new object file format; see *old*(8) for information on dealing with old format programs.

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.

−g Have the compiler produce additional symbol table information for *sdb*(1). Also pass the **−lg** flag to *ld*(1).

−w Suppress warning diagnostics.

−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 Invoke an object-code improver.

−R Passed on to *as*, making initialized variables shared and read-only.

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

−E Run only the macro preprocessor on the named C programs, and send the result to the standard output.

−C prevent the macro preprocessor from eliding comments.

−o *output*
Name the final 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 directories on a standard list.

−B*string*
Find substitute compiler passes in the files named *string* with the suffixes cpp, ccom and c2. If *string* is empty, use a standard backup version.

−t[p012]
Find only the designated compiler passes in the files whose names are constructed by a **−B** option. In the absence of a **−B** option, the *string* is taken to be '/usr/c/'.

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

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporary
/lib/cpp	preprocessor
/lib/ccom	compiler
/usr/c/occom	backup compiler
/usr/c/ocpp	backup preprocessor
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library, see (3)
/usr/include	standard directory for '#include' files

SEE ALSO

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*
monitor(3), prof(1), adb(1), ld(1), sdb(1), old(8)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader.

BUGS

The compiler currently ignores advice to put **char, unsigned char, short** or **unsigned short** variables in registers. It previously produced poor, and in some cases incorrect, code for such declarations.

NAME

cd − change working directory

SYNOPSIS

cd directory

DESCRIPTION

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

Because a new process is created to execute each command, *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).

SEE ALSO

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

NAME
checknr − check nroff/troff files

SYNOPSIS
checknr −s −f [**−a**.*x1.y1.x2.y2. ... .xn.yn*] *file* ...

DESCRIPTION
Checknr checks a list of *nroff*(1) or *troff*(1) input files for certain kinds of errors involving mismatched opening and closing delimiters and unknown commands. Delimeters checked are:

(1) Font changes using \fx ... \fP.

(2) Size changes using \sx ... \s0.

(3) Macros that come in open ... close forms, for example, the .TS and .TE macros which must always come in pairs.

Checknr knows about the *ms*(7) and *me*(7) macro packages.

Additional pairs of macros can be added to the list using the **−a** option. This must be followed by groups of six characters, each group defining a pair of macros. The six characters are a period, the first macro name, another period, and the second macro name. For example, to define a pair .BS and .ES, use **−a**.BS.ES

The **−f** option requests *checknr* to ignore \f font changes.

The **−s** option requests *checknr* to ignore \s size changes.

Checknr is intended to be used on documents that are prepared with *checknr* in mind, much the same as *lint*. It expects a certain document writing style for \f and \s commands, in that each \fx must be terminated with \fP and each \sx must be terminated with \s0. While it will work to directly go into the next font or explicitly specify the original font or point size, and many existing documents actually do this, such a practice will produce complaints from *checknr*. Since it is probably better to use the \fP and \s0 forms anyway, you should think of this as a contribution to your document preparation style.

SEE ALSO
nroff(1), troff(1), ms(7), me(7), checkeqn(1)

DIAGNOSTICS
Complaints about unmatched delimiters.
Complaints about unrecognized commands.
Various complaints about the syntax of commands.

AUTHOR
Mark Horton

BUGS
There is no way to define a 1 character macro name using **−a**

NAME

chfn — change full name of user

SYNOPSIS

chfn name string

DESCRIPTION

Chfn is a command similar to *passwd*(1) except that it is used to change the gcos field of the password file rather than the password entry. Note that the string specified will replace the entire gecos field of the specified user. If (as on the UCB system) this field contains information in addition to the user's full name, this information must be included in string or it will be deleted. Hence chfn can be used to fix phone numbers, offices, etc.

An example use of this command would be

chfn mark '& Horton,508E,7686,5240633'

Note that the string must in general be quoted to shield blanks and special characters from the shell. The field should consist of the users name, followed by their office number, followed by the last 4 digits of their office extension and finally their home phone number. Any of these can be omitted. At UCB, offices can be given as "508E" for 508 Evans, and "187MC" for 187M Cory. There should be no spaces in the entry except for those in your name.

It is a good idea to run *finger*(1) on the user before and after *chfn* to make sure you have formatted the data correctly.

SEE ALSO

finger(1), passwd(5), chsh(1), passwd(1)

AUTHOR

Mark Horton

BUGS

The encoding of the office and extension information is installation dependent.

A new user-information data base is in the works which will supplant this program; *chfn's* days are numbered.

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 of the following modes:

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

A symbolic *mode* has the form:

[*who*] *op permission* [*op permission*] ...

The *who* part is a combination of the letters **u** (for user's permissions), **g** (group) and **o** (other). The letter **a** stands for **ugo.** If *who* is omitted, the default is *a* but the setting of the file creation mask (see umask(2)) is taken into account.

Op can be **+** to add *permission* to the file's mode, **−** to take away *permission* and **=** to assign *permission* absolutely (all other bits will be reset).

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

The first example denies write permission to others, the second makes a file executable:

```
chmod o−w file
chmod +x file
```

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

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

SEE ALSO

ls(1), chmod(2), stat(2), umask(2), chown(8)

NAME

chsh – change default login shell

SYNOPSIS

chsh name [shell]

DESCRIPTION

Chsh is a command similar to *passwd*(1) except that it is used to change the login shell field of the password file rather than the password entry. If no *shell* is specified then the shell reverts to the default login shell */bin/sh.* Otherwise only */bin/csh* or */bin/oldcsh* can be specified as the shell unless you are the super-user.

An example use of this command would be

```
chsh bill /bin/csh
```

SEE ALSO

csh(1), passwd(1), passwd(5)

NAME

cifplot — CIF interpreter and plotter

SYNOPSIS

cifplot [*options*] file1.cif [file2.cif ...]

DESCRIPTION

Cifplot takes a description in Cal-Tech Intermediate Form (CIF) and produces a plot. CIF is a low-level graphics language suitable for describing integrated circuit layouts. Although CIF can be used for other graphics applications, for ease of discussion it will be assumed that CIF is used to describe integrated circuit designs. *Cifplot* interprets any legal CIF 2.0 description including symbol renaming and Delete Definition commands. In addition, a number of local extensions have been added to CIF, including text on plots and include files. These are discussed later. Care has been taken to avoid any arbitrary restrictions on the CIF programs that can be plotted.

To get a plot call *cifplot* with the name of the CIF file to be plotted. If the CIF description is divided among several files call *cifplot* with the names of all files to be used. *Cifplot* reads the CIF description from the files in the order that they appear on the command line. Therefore the CIF *End* command should be only in the last file since *cifplot* ignores everything after the *End* command. After reading the CIF description but before plotting, *cifplot* will print a estimate of the size of the plot and then ask if it should continue to produce a plot. Type **y** to proceed and **n** to abort. A typical run might look as follows:

```
% cifplot lib.cif sorter.cif
Window -5700 174000 -76500 168900
Scale: 1 micron is 0.004075 inches
The plot will be 0.610833 feet
Do you want a plot?  y
```

After typing *y* *cifplot* will produce a plot on the Benson-Varian plotter.

Cifplot recognizes several command line options. These can be used to change the size and scale of the plot, change default plot options, and to select the output device. Several options may be selected. A dash(-) must precede each option specifier. The following is a list of options that may be included on the command line:

−w *xmin xmax ymin ymax*
: (**window**) The -w options specifies the window; by default the window is set to be large enough to contain the entire plot. The windowing commands lets you plot just a small section of your chip, enabling you to see it in better detail. *Xmin, xmax, ymin,* and *ymax* should be specified in CIF coordinates.

−s *float*
: (**scale**) The -s option sets the scale of the plot. By default the scale is set so that the window will fill the whole page. *Float* is a floating point number specifying the number of inches which represents 1 micron. A recommended size is 0.02.

−l *layer_list*
: (**layer**) Normally all layers are plotted. The -l option specifies which layers NOT to plot. The *layer_list* consists of the layer names separated by commas, no spaces. There are two reserved names: **text** and **bbox**. Including the layer name **text** in the list suppresses the plotting of text; **bbox** suppresses the bounding box around symbols.

−d *n* (**depth**) This option lets you limit the amount of detail plotted in a hierarchically designed chip. It will only instanciate the plot down *n* levels of calls. Sometimes too much detail can hide important features in a circuit.

−g *n* (**grid**) Draw a grid over the plot with spacing every *n* CIF units.

−f (**fuzzy**) Don't print the border outlines around the merged features on each layer.

−h (**half**) Plot at half normal resolution. *(Not yet implemented.)*

−e (**extensions**) Accept only standard CIF. User extensions produce warnings.

−I (**non-Interactive**) Do not ask for confirmation. Always plot.

−L (**List**) Produce a listing of the CIF file on standard output as it is parsed. Not recommended unless debugging hand-coded CIF since CIF code can be rather long.

−a *n* (**approximate**) Approximate a roundflash with an *n*-sided polygon. By default *n* equals 8. (I.e. roundflashes are approximated by octagons.) If *n* equals 0 then output circles for roundflashes. (It is best not to use full circles since they significantly slow down plotting.) *(Full circles not yet implemented.)*

−b "*text*"
(**banner**) Print the text at the top of the plot.

−C (**Comments**) Treat comments as though they were spaces. Sometimes CIF files created at other universities will have several errors due to syntactically incorrect comments. (I.e. the comments may appear in the middle of a CIF command or the comment does not end with a semi-colon.) Of course, CIF files should not have any errors and these comment related errors must be fixed before transmitting the file for fabrication. But many times fixing these errors seems to be more trouble than it is worth, especially if you just want to get a plot. This option is useful in getting rid of many of these comment related syntax errors.

−r (**rotate**) Rotate the plot 90 degrees.

−V (**Varian**) Send output to the varian. (This is the default option.)

−W (**Wide**) Send output directly to the versatec.

−S (**Spool**) Store the output in a temporary file then dump the output quickly onto the Versatec. Makes nice crisp plots; also takes up a lot of disk space.

−T *n* (**Terminal**) Send output byte stream to standard output. Useful for setting up pipes. *N* is the number of bytes per line that the plotting device expects. *(Not yet implemented.)*

−H (**HP2648**) Send output to HP2648A terminal. This requires that *cifplot* is running in the foreground on an HP2648 and that there is a scratch tape in the left tape drive of the terminal. *(Not yet fully implemented.)*

−X *basename*
(**eXtractor**) From the CIF file create a circuit description suitable for switch level simulation. It creates two files: *basename*.**sim** which contains the circuit description, and *basename*.**node** which contains the node numbers and their location used in the circuit description.

When this option is invoked no plot is made. Therefore it is advisable not to use any of the other options that deal only with plotting. However, the *window*, *layer*, and *approximate* options are still appropriate. To get a plot of the circuit with the node numbers call *cifplot* again, without the −X option, and include *basename*.**nodes** in the list of CIF files to be plotted. (This file must appear in the list of files before the file with the CIF End command.) *(Not yet fully implemented.)*

−P *pattern_file*
(**Pattern**) The -P option lets you specify your own layers and stipple patterns. *Pattern_file* may contain an arbitrary number of layer descriptors. A layer descriptor is the layer name in double quotes, followed by 8 integers. Each integer specifies 32 bits where ones are black and zeroes are white. Thus the 8 integers specify a 32 by 8 bit

stipple pattern. The integers may be in decimal, octal, or hex. Hex numbers start with '0x'; octal numbers start with '0'. The CIF syntax requires that layer names be made up of only uppercase letters and digits, and not longer than four characters. The following is example of a layer description for poly-silicon:

```
"NP"    0x08080808 0x04040404 0x02020202 0x01010101
        0x80808080 0x40404040 0x20202020 0x10101010
```

−F *font_file*
: **(Font)** The -F option indicates which font you want for your text. The file must be in the directory '/usr/lib/vfont'. The default font is Roman 6 point. Obviously, this option is only useful if you have text on your plot.

−O *filename*
: **(Output)** After parsing the CIF files, store an equivalent but easy to parse CIF description in the specified file. This option removes the include and array commands (see next section) and replaces them with equivalent standard CIF statements. The resulting file is suitable for transmission to other facilities for fabrication.

In the definition of CIF provisions were made for local extensions. All extension commands begin with a number. Part of the purpose of these extensions is to test what features would be suitable to include as part of the standard language. But it is important to realize that these extensions are not standard CIF and that many programs interpreting CIF do not recognize them. If you use these extensions it is advisable to create another CIF file using the -O options described above before submitting your circuit for fabrication. The following is a list of extensions recognized by *cifplot*.

0I *filename*;
: **(Include)** Read from the specified file as though it appeared in place of this command. Include files can be nested up to 6 deep.

0A *s m n dx dy* ;
: **(Array)** Repeat symbol *s m* times with *dx* spacing in the x-direction and *n* times with *dy* spacing in the y-direction. *s*, *m*, and *n* are unsigned integers. *dx* and *dy* are signed integers in CIF units.

1 *message*;
: **(Print)** Print out the message on standard output when it is read.

2 "*text*" *transform* ;

2C "*text*" *transform* ;
: **(Text on Plot)** *Text* is placed on the plot at the position specified by the transformation. The allowed transformations are the same as the those allowed for the Call command. The transformation affects only the point at which the beginning of the text is to appear. The text is always plotted horizontally, thus the mirror and rotate transformations are not really of much use. Normally text is placed above and to the right of the reference point. The 2C command centers the text about the reference point.

9 *name*;
: **(Name symbol)** *name* is associated with the current symbol.

94 *name x y*;

94 *name x y layer*;
: **(Name point)** *name* is associated with the point (*x*, *y*). Any mask geometry crossing this point is also associated with *name*. If *layer* is present then just geometry crossing the point on that layer is associated with *name*. For plotting this command is similar to text on plot. When doing circuit extraction this command is used to give an explicit name to a node. *Name* must not have any spaces in it, and it should not be a number.

FILES

/usr/lib/vdump
/usr/lib/vfont/*
/usr/lib/vpd
/usr/tmp/cif*

ALSO SEE

A Guide to LSI Implementation by Hon and Sequin, Second Edition (Xerox PARC, 1980) for a description of CIF.
Introduction to VLSI Systems by Mead and Conway (Addison-Wesley, 1980)

AUTHOR

Dan Fitzpatrick

BUGS

Output should be spooled.

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.

FILES

/etc/termcap terminal capability data base

BUGS

NAME

cmp − compare two files

SYNOPSIS

cmp [**−l**] [**−s**] file1 file2

DESCRIPTION

The two files are compared. (If *file1* is '−', the standard input is used.) Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

Options:

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

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

SEE ALSO

diff(1), comm(1)

DIAGNOSTICS

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

NAME

col − filter reverse line feeds

SYNOPSIS

col [−bfx]

DESCRIPTION

Col reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ESC-7 in ASCII) and by forward and reverse half line feeds (ESC-9 and ESC-8). *Col* is particularly useful for filtering multicolumn output made with the '.rt' command of *nroff* 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.

SEE ALSO

troff(1), tbl(1), greek(1)

BUGS

Can't back up more than 128 lines.
No more than 800 characters, including backspaces, on a line.

NAME

colcrt – filter nroff output for CRT previewing

SYNOPSIS

colcrt [**–**] [**–2**] [file ...]

DESCRIPTION

Colcrt provides virtual half-line and reverse line feed sequences for terminals without such capability, and on which overstriking is destructive. Half-line characters and underlining (changed to dashing '–') are placed on new lines in between the normal output lines.

The optional **–** suppresses all underlining. It is especially useful for previewing *allboxed* tables from *tbl*(1).

The option **–2** causes all half-lines to be printed, effectively double spacing the output. Normally, a minimal space output format is used which will suppress empty lines. The program never suppresses two consecutive empty lines, however. The **–2** option is useful for sending output to the line printer when the output contains superscripts and subscripts which would otherwise be invisible.

A typical use of *colcrt* would be

    tbl exum2.n | nroff –ms | colcrt – | more

SEE ALSO

nroff/troff(1), col(1), more(1), ul(1)

AUTHOR

William Joy

BUGS

Should fold underlines onto blanks even with the '–' option so that a true underline character would show; if we did this, however, *colcrt* wouldn't get rid of *cu'd* underlining completely.

Can't back up more than 102 lines.

General overstriking is lost; as a special case '|' overstruck with '–' or underline becomes '+'.

Lines are trimmed to 132 characters.

Some provision should be made for processing superscripts and subscripts in documents which are already double-spaced.

NAME

colrm − remove columns from a file

SYNOPSIS

colrm [startcol [endcol]]

DESCRIPTION

Colrm removes selected columns from a file. Input is taken from standard input. Output is sent to standard output.

If called with one parameter the columns of each line will be removed starting with the specified column. If called with two parameters the columns from the first column to the last column will be removed.

Column numbering starts with column 1.

SEE ALSO

expand(1)

AUTHOR

Jeff Schriebman

BUGS

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.

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

SEE ALSO

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

NAME

compact, uncompact, ccat − compress and uncompress files, and cat them

SYNOPSIS

compact [name ...]
uncompact [name ...]
ccat [file ...]

DESCRIPTION

Compact compresses the named files using an adaptive Huffman code. If no file names are given, then the standard input is compacted to the standard output. *Compact* operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to prepend a decoding tree to the compressed file since the encoder and the decoder start in the same state and stay synchronized. Furthermore, *compact* and *uncompact* can operate as filters. In particular,

```
... | compact | uncompact | ...
```

operates as a (very slow) no-op.

When an argument *file* is given, it is compacted and the resulting file is placed in *file.C; file* is unlinked. The first two bytes of the compacted file code the fact that the file is compacted. This code is used to prohibit recompaction.

The amount of compression to be expected depends on the type of file being compressed. Typical values of compression are: Text (38%), Pascal Source (43%), C Source (36%) and Binary (19%). These values are the percentages of file bytes reduced.

Uncompact restores the original file from a file compressed by *compact.* If no file names are given, then the standard input is uncompacted to the standard output.

Ccat cats the original file from a file compressed by *compact,* without uncompressing the file.

RESTRICTION

The last segment of the filename must contain fewer than thirteen characters to allow space for the appended '.C'.

FILES

*.C	compacted file created by compact, removed by uncompact

SEE ALSO

Gallager, Robert G., "Variations on a Theme of Huffman", *I.E.E.E. Transactions on Information Theory,* vol. IT-24, no. 6, November 1978, pp. 668 - 674.

AUTHOR

Colin L. Mc Master

BUGS

NAME

cp — copy

SYNOPSIS

cp [**−i**] file1 file2

cp [**−i**] 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.

If the **−i** option is specified, *cp* will prompt the user with the name of the file whenever the copy will cause an old file to be overwritten. An answer of 'y' will cause *cp* to continue. Any other answer will prevent it from overwriting the file.

SEE ALSO

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

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

SEE ALSO

ed(1), makekey(8)

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, Bell Telephone Laboratories assumes no responsibility for their use by the recipient. Further, Bell Laboratories assumes no 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 first implementation of a command language interpreter incorporating a history mechanism (see **History Substitutions**) job control facilities (see **Jobs**) and a C-like syntax. So as to be able to use its job control facilities, users of *csh* must (and automatically) use the new tty driver summarized in *newtty*(4) and fully described in *tty*(4). This new tty driver allows generation of interrupt characters from the keyboard to tell jobs to stop. See *stty*(1) for details on setting options in the new tty driver.

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 users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters ‘&’ ‘|’ ‘;’ ‘<’ ‘>’ ‘(’ ‘)’ form separate words. If doubled in ‘&&’, ‘||’, ‘<<’ or ‘>>’ these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with ‘\’. A newline preceded by a ‘\’ is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, ‘`’, ‘'’ or ‘"’, form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of ‘`’ or ‘"’ characters a newline preceded by a ‘\’ gives a true newline character.

When the shell's input is not a terminal, the character ‘#’ introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by ‘\’ and in quotations using ‘'’, ‘`’, and ‘"’.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by ‘|’ characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by ‘;’, and are then executed sequentially. A sequence of pipelines may be executed without immediately waiting for it to terminate by following it with an ‘&’.

Any of the above may be placed in ‘(’ ‘)’ to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with ‘||’ or ‘&&’ indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See *Expressions.)*

Jobs

The shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the *jobs* command, and assigns them small integer numbers. When a job is started asynchronously with '&', the shell prints a line which looks like:

[1] 1234

indicating that the jobs which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you may hit the key **^Z** (control-Z) which sends a STOP signal to the current job. The shell will then normally indicate that the job has been 'Stopped', and print another prompt. You can then manipulate the state of this job, putting it in the background with the *bg* command, or run some other commands and then eventually bring the job back into the foreground with the foreground command *fg*. A **^Z** takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is another special key ^Y which does not generate a STOP signal until a program attempts to *read*(2) it. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command "stty tostop". If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character '%' introduces a job name. If you wish to refer to job number 1, you can name it as '%1'. Just naming a job brings it to the foreground; thus '%1' is a synonym for 'fg %1', bringing job 1 back into the foreground. Similarly saying '%1 &' resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous, thus '%ex' would normally restart a suspended *ex*(1) job, if there were only one suspended job whose name began with the string 'ex'. It is also possible to say '%?string' which specifies a job whose text contains *string*, if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a '+' and the previous job with a '−'. The abbreviation '%+' refers to the current job and '%−' refers to the previous job. For close analogy with the syntax of the *history* mechanism (described below), '%%' is also a synonym for the current job.

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. If, however, you set the shell variable *notify*, the shell will notify you immediately of changes of status in background jobs. There is also a shell command *notify* which marks a single process so that its status changes will be immediately reported. By default *notify* marks the current process; simply say 'notify' after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that 'You have stopped jobs.' You may use the *jobs* command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

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 prevent its special meaning; for convenience, a '!' is passed unchanged when it is followed by a blank, tab, newline, '=' or '('. (History substitutions also occur when an input line begins with '↑'. This special abbreviation will be described later.) Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The size of which is controlled by the *history* variable; the previous command is always retained, regardless of its value. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the *history* command:

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

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an '!' in the prompt string.

With the current event 13 we can refer to previous events by event number '!11', relatively as in '!−2' (referring to the same event), by a prefix of a command word as in '!d' for event 12 or '!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.*

To select words from an event we can follow the event specification by a ':' and a designator for the desired words. The words of a input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	*n*'th argument
↑	first argument, i.e. '1'
$	last argument
%	word matched by (immediately preceding) ?*s*? search
x−*y*	range of words
−*y*	abbreviates '0−*y*'
*	abbreviates '↑−$', or nothing if only 1 word in event
*x**	abbreviates '*x*−$'
x−	like '*x**' but omitting word '$'

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

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing '.xxx' component, leaving the root name.
e	Remove all but the extension '.xxx' part.
s/*l*/*r*/	Substitute *l* for *r*
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 a newline follows immediately as may the trailing '?' in a contextual scan.

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

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a '↑'. This is equivalent to '!:s↑' providing a convenient shorthand for substitutions on the text of the previous line. Thus '↑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. Strings enclosed in ''' are prevented any further interpretation. Strings enclosed in '"' are yet variable and command expanded as described below.

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

Alias substitution

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

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

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

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

Variable substitution

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

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

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

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

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

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

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

$name
${name}
    Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters 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 *home*. When followed by a name consisting of letters, digits and '−' characters the shell searches for a user with that name and substitutes their home directory; thus '~ken' might expand to '/usr/ken' and '~ken/chmach' to '/usr/ken/chmach'. If the character '~' is followed by a character other than a letter or '/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation 'a{b,c,d}e' is a shorthand for 'abe ace ade'. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus '~source/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, its is truncated, its previous contents being lost.

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

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

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

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

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. Note that the default standard input for a command run detached is **not** modified to be the empty file '/dev/null'; rather the standard input remains as the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see **Jobs** above.)

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.*

alloc
    Shows the amount of dynamic core in use, broken down into used and free core, and address of the last location in the heap. With an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

bg
bg %job ...
    Puts the current or specified jobs into the background, continuing them if they were stopped.

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.

dirs
Prints the directory stack; the top of the stack is at the left, the first directory in the stack being the current directory.

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.

eval arg ...
(As in *sh*(1).) The arguments are read as input to the shell and the resulting command(s) executed. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See *tset*(1) for an example of using *eval*.

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).

fg
fg %job ...
Brings the current or specified jobs into the foreground, continuing them if they were stopped.

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.

hashstat

Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding *exec's).* An *exec* is attempted for each component of the *path* where the hash function indicates a possible hit, and in each component which does not begin with a '/'.

history
history *n*
history −r *n*

Displays the history event list; if *n* is given only the *n* most recent events are printed. The −r option reverses the order of printout to be most recent first rather than oldest first.

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.)*

jobs

jobs −l

Lists the active jobs; given the −l options lists process id's in addition to the normal information.

kill %job
kill −sig %job ...
kill pid
kill −sig pid ...
kill −l

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or 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 job. 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.

limit
limit *resource*
limit *resource maximum-use*

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource.* If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given.

Resources controllable currently include *cputime* (the maximum number of cpu-seconds to be used by each process), *filesize* (the largest single file which can be created), *datasize* (the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text), *stacksize* (the maximum size of the automatically-extended stack region), and *coredumpsize* (the size of the largest core dump that will be created).

The *maximum-use* may be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is 'k' or 'kilobytes' (1024 bytes); a scale factor of 'm' or 'megabytes' may also be used. For *cputime* the default scaling is 'seconds', while 'm' for minutes or 'h' for hours, or a time of the form 'mm:ss' giving minutes and seconds may be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

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.

newgrp

Changes the group identification of the caller; for details see *newgrp*(1). A new shell is executed by *newgrp* so that the shell state is lost.

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.*

notify
notify %job ...

Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

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.

popd
popd +n

Pops the directory stack, returning to the new top directory. With a argument '+*n*' discards the *n*th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top.

pushd
pushd name
pushd +n

With no arguments, *pushd* exchanges the top two elements of the directory stack. Given a *name* argument, *pushd* changes to the new directory (ala *cd)* and pushes the old current working directory (as in *csw)* onto the directory stack. With a numeric argument, rotates the *n*th argument of the directory stack around to be the top element and changes to it. The members of the directory stack are numbered from the top starting at 0.

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 most commonly used environment variable USER, TERM, and PATH are automatically imported to and exported from the *csh* variables *user, term,* and *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.

stop
stop %job ...

Stops the current or specified job which is executing in the background.

suspend

Causes the shell to stop in its tracks, much as if it had been sent a stop signal with ^Z. This is most often used to stop shells started by *su*(1).

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.

unlimit *resource*
unlimit

Removes the limitation on *resource.* If no *resource* is specified, then all *resource* limitations are removed.

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.*

unsetenv pattern

Removes all variables whose name match the specified pattern from the environment. See also the *setenv* command above and *printenv*(1).

wait

All background jobs are waited for. It 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.

%job

Brings the specified job into the foreground.

%job &

Continues the specified job in the background.

@
@ 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, cwd, home, path, prompt, shell* and *status* are always set by the shell. Except for *cwd* and *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. (It could be set once in the *.login* except that commands through *net*(1) would not see the definition.)

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.

cwd The full pathname of the current directory.

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

histchars Can be given a string value to change the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character !. The second character of its value replaces the character ↑ in quick substitutions.

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.

notify If set, the shell notifies asynchronously of job completions. The default is to rather present job completions just before printing a prompt.

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

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

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

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

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

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

Non-builtin command execution

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

'/', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

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

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

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

Argument list processing

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

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

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

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

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

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

−s Command input is taken from the standard input.

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

−v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.

−x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.

−V Causes the *verbose* variable to be set even before '.cshrc' is executed.

−X Is to **−x** as **−V** is to **−v.**

After processing of flag arguments if arguments remain but none of the **−c, −i, −s,** or **−t** options was given the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by '$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a 'standard' shell if the first character of a script is not a '#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable *argv.*

Signal handling

The shell normally ignores *quit* signals. Jobs running detached (either by '&' or the *bg* or %... & commands) 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. Job control and directory stack features first implemented by J.E. Kulp of I.I.A.S.A, Laxenburg, Austria, with different syntax than that used now.

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 10240 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* substitiutions on a single line to 20.

SEE ALSO

sh(1), newcsh(1), access(2), exec(2), fork(2), killpg(2), pipe(2), sigsys(2), umask(2), vlimit(2), wait(2), jobs(3), sigset(3), tty(4), a.out(5), environ(5), 'An introduction to the C shell'

BUGS

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (i.e. wrong) as the job may have changed directories internally.

Shell builtin functions are not stoppable/restartable. Command sequences of the form 'a ; b ; c' are also not handled gracefully when stopping is attempted. If you suspend 'b', the shell will then immediately execute 'c'. This is especially noticeable if this expansion results from an *alias*. 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 – create a tags file

SYNOPSIS

ctags [**−u**] [**−v**] [**−w**] [**−x**] name ...

DESCRIPTION

Ctags makes a tags file for *ex*(1) from the specified C, Pascal and Fortran sources. A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *ex* can quickly find these function definitions.

If the **−x** flag is given, *ctags* produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. This is a simple index which can be printed out as an off-line readable function index.

If the **−v** flag is given, an index of the form expected by *vgrind*(1) is produced on the standard output. This listing contains the function name, file name, and page number (assuming 64 line pages). Since the output will be sorted into lexicographic order, it may be desired to run the output through **sort −f**. Sample use:

```
ctags −v files | sort −f > index
vgrind −x index
```

Files whose name ends in **.c** or **.h** are assumed to be C source files and are searched for C routine and macro definitions. Others are first examined to see if they contain any Pascal or Fortran routine definitions; if not, they are processed again looking for C definitions.

Other options are:

−w suppressing warning diagnostics.

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

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

FILES

tags output tags file

SEE ALSO

ex(1), vi(1)

AUTHOR

Ken Arnold; FORTRAN added by Jim Kleckner; Bill Joy added Pascal and **−x**, replacing *cxref*.

BUGS

Recognition of **functions, subroutines** and **procedures** for FORTRAN and Pascal is done is a very simpleminded way. No attempt is made to deal with block structure; if you have two Pascal procedures in different blocks with the same name you lose.

The method of deciding whether to look for C or Pascal and FORTRAN functions is a hack.

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 [yymmddhhmm [.ss]]

DESCRIPTION

If no argument is given, the current date and time are printed. If an argument is given, the current date is set. *yy* is the last two digits of the year; the first *mm* is the month number; *dd* is the day number in the month; *hh* is the hour number (24 hour system); the second *mm* is the minute number; *.ss* is optional and is the seconds. For example:

date 10080045

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

FILES

/usr/adm/wtmp to record time-setting

SEE ALSO

utmp(5)

DIAGNOSTICS

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

BUGS

The system attempts to keep the date in a format closely compatible with VMS. VMS, however, uses local time (rather than GMT) and does not understand daylight savings time. Thus if you use both UNIX and VMS, VMS will be running on GMT.

NAME

dc − desk calculator

SYNOPSIS

dc [file]

DESCRIPTION

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

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

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

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

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

d
: The top value on the stack is duplicated.

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

; : are used by *bc* for array operations.

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

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

SEE ALSO

bc(1), which is a preprocessor for *dc* providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

'x is unimplemented' where x is an octal number.
'stack empty' for not enough elements on the stack to do what was asked.
'Out of space' when the free list is exhausted (too many digits).
'Out of headers' for too many numbers being kept around.
'Out of pushdown' for too many items on the stack.
'Nesting Depth' for too many levels of nested execution.

NAME

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.

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

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

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

SEE ALSO

cp(1), tr(1)

DIAGNOSTICS

f+p records in(out): numbers of full and partial records 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

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.

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 [**−i**] [**−l**] [filesystem ...] [file ...]

DESCRIPTION

Df prints out the number of free blocks available on the specified *filesystem,* e.g. "/dev/rp0a", or on the filesystem in which the specified *file,* e.g. "$HOME" in contained. If no file system is specified, the free space on all of the normally mounted file systems is printed.

The reported numbers are in file system block units; currently each filesystem block is 1024 bytes long, twice the size of the blocks reported by *du*(1) or *ls*(1) with the **−s** option.

Other options are:

−i Report also the number of inodes which are used and free.

−l examines also the free list, double checking that the summary number in the filesystem superblock is correct.

FILES

/etc/fstab list of normally mounted filesystems

SEE ALSO

fstab(5), icheck(8), quot(8)

NAME

diction,explain − print wordy sentences; thesaurus for diction

SYNOPSIS

diction [**−ml**] [**−mm**] [**−n**] [**−f** pfile] file ...
explain

DESCRIPTION

Diction finds all sentences in a document that contain phrases from a data base of bad or wordy diction. Each phrase is bracketed with [ ]. Because *diction* runs *deroff* before looking at the text, formatting header files should be included as part of the input. The default macro package **−ms** may be overridden with the flag **−mm.** The flag **−ml** which causes **deroff** to skip lists, should be used if the document contains many lists of non-sentences. The user may supply her/his own pattern file to be used in addition to the default file with **−f** *pfile.* If the flag **−n** is also supplied the default file will be suppressed.

Explain is an interactive thesaurus for the phrases found by diction.

SEE ALSO

deroff(1)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks. In particular, *diction* doesn't grok **−me.**

NAME

diff − differential file and directory comparator

SYNOPSIS

diff [**−l**] [**−r**] [**−s**] [**−cefh**] [**−b**] dir1 dir2
diff [**−cefh**] [**−b**] file1 file2
diff [**−D***string*] [**−b**] file1 file2

DESCRIPTION

If both arguments are directories, *diff* sorts the contents of the directories by name, and then runs the regular file *diff* algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed. Options when comparing directories are:

−l long output format; each text file *diff* is piped through *pr*(1) to paginate it, other differences are remembered and summarized after all text file differences are reported.

−r causes application of *diff* recursively to common subdirectories encountered.

−s causes *diff* to report files which are the same, which are otherwise not mentioned.

−Sname
starts a directory *diff* in the middle beginning with file *name.*

When run on regular files, and when comparing text files which differ during directory comparison, *diff* tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, *diff* finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, then either may be given as '−', in which case the standard input is used. If *file1* is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* is used (and vice versa).

There are several options for output format; the default output format contains lines of these forms:

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

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

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

Except for **−b**, which may be given with any of the others, the following options are mutually exclusive:

−e producing a script of *a, c* and *d* commands for the editor *ed,* which will recreate *file2* from *file1.* 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
```

Extra commands are added to the output when comparing directories with **−e**, so that the result is a *sh*(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2.*

−f produces a script similar to that of **−e**, not useful with *ed,* and in the opposite order.

−c produces a diff with lines of context. The default is to present 3 lines of context and may be changed, e.g to 10, by **−c10**. With **−c** the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with '−'; those added to *file2* are marked '+'. Lines which are changed from one file to the other are marked in both files with '!'.

−h does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length.

−Dstring
causes *diff* to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1*, while defining *string* will yield *file2*.

−b causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

FILES

/tmp/d?????
/usr/lib/diffh for **−h**
/usr/bin/pr

SEE ALSO

cmp(1), cc(1), comm(1), ed(1), diff3(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 '.'.

When comparing directories with the **−b** option specified, *diff* first compares the files ala *cmp*, and then decides to run the *diff* algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical because the only differences are insignificant blank string differences.

NAME

diff3 – 3-way differential file comparison

SYNOPSIS

diff3 [**−ex3**] file1 file2 file3

DESCRIPTION

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

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

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

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **−e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, *i.e.* the changes that normally would be flagged ==== and ====3. Option **−x** (**−3**) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to 'file1'.

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

FILES

/tmp/d3?????
/usr/lib/diff3

SEE ALSO

diff(1)

BUGS

Text lines that consist of a single '.' will defeat **−e**.
Files longer than 64K bytes won't work.

NAME

du − summarize disk usage

SYNOPSIS

du [**−s**] [**−a**] [name ...]

DESCRIPTION

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

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

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

SEE ALSO

df(1), quot(8)

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 multiply.

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'.

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. The special form '%' is an abbreviation for the address pair '1,$'.

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.

(. , .) t *a*
: This command acts just like the *m* command, except that a copy of the addressed lines is placed after address *a* (which may be 0). '.' is left on the last line of the copy.

(. , .) u
: The undo command restores the buffer to it's state before the most recent buffer modifying command. The current line is also restored. Buffer modifying commands are *a, c, d, g, i, k,* and *v*. For purposes of undo, *g* and *v* are considered to be a single buffer modifying command. Undo is its own inverse.

: When *ed* runs out of memory (at about 8000 lines on any 16 bit mini-computer such as the PDP-11) This full undo is not possible, and *u* can only undo the effect of the most recent substitute on the current line. This restricted unto also applies to editor scripts when *ed* is invoked with the - option.

(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.

(1, $) wq filename
: This command is the same as *w* except that afterwards a *q* command is done, exiting the editor after the file is written.

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. (. +1) z or,

(. +1) z*n*
: This command scrolls through the buffer starting at the addressed line. 22 (or *n*, if given) lines are printed. The last line printed becomes the current line. The value *n* is sticky, in that it becomes the default for future *z* commands.

($) =
: 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

SEE ALSO

B. W. Kernighan, *A Tutorial Introduction to the ED Text Editor*
B. W. Kernighan, *Advanced editing on UNIX*
sed(1), crypt(1)

DIAGNOSTICS

'?name' for inaccessible file; '?self-explanitory 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.
The *x* command, **-x** option, and special treatment of hangups only work on UNIX.

NAME

efl − Extended Fortran Language

SYNOPSIS

efl [option ...] [filename ...]

DESCRIPTION

Efl compiles a program written in the EFL language into clean Fortran. *Efl* provides the same control flow constructs as does Ratfor (1), which are essentially identical to those in C:

statement grouping with braces;

decision-making with if, if-else, and switch-case; while, for, Fortran do, repeat, and repeat...until loops; multi-level break and next. In addition, EFL has C-like data structures, and more uniform and convenient input/output syntax, generic functions. EFL also provides some syntactic sugar to make programs easier to read and write:

free form input:

multiple statements/line; automatic continuation statement label names (not just numbers),

comments:

this is a comment

translation of relationals:

>, >=, etc., become .GT., .GE., etc.

return (expression)

returns expression to caller from function

define: define name replacement

include:

include filename

The Efl command option **−w** suppresses warning messages. The option **−C** causes comments to be copied through to the Fortran output (default); **−#** prevents comments from being copied through. If a command argument contains an embedded equal sign, that argument is treated as if it had appeared in an **option** statement at the beginning of the program. *Efl* is best used with *f77*(1).

SEE ALSO

f77(1), ratfor(1).

S. I. Feldman, *The Programming Language EFL*, Bell Labs Computing Science Technical Report #78.

NAME

eqn, neqn, checkeq — typeset mathematics

SYNOPSIS

eqn [**−dxy**] [**−pn**] [**−sn**] [**−fn**] [file] ...
checkeq [file] ...

DESCRIPTION

Eqn is a troff(1) preprocessor for typesetting mathematics on a Graphic Systems phototypesetter, *neqn* on terminals. Usage is almost always

```
eqn file ... | troff
neqn file ... | nroff
```

If no files are specified, these programs reads from the standard input. A line beginning with '.EQ' marks the start of an equation; the end of an equation is marked by a line beginning with '.EN'. Neither of these lines is altered, so they may be defined in macro packages to get centering, numbering, etc. It is also possible to set two characters as 'delimiters'; subsequent text between delimiters is also treated as *eqn* input. Delimiters may be set to characters *x* and *y* with the command-line argument **−d***xy* or (more commonly) with 'delim *xy*' between .EQ and .EN. The left and right delimiters may be identical. Delimiters are turned off by 'delim off'. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

The program *checkeq* reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within *eqn* are separated by spaces, tabs, newlines, braces, double quotes, tildes or circumflexes. Braces {} are used for grouping; generally speaking, anywhere a single character like *x* could appear, a complicated construction enclosed in braces may be used instead. Tilde ~ represents a full space in the output, circumflex ^ half as much.

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* yields $\frac{a}{b}$.

sqrt makes square roots: *1 over sqrt {ax sup 2 +bx+c}* results in $\frac{1}{\sqrt{ax^2+bx+c}}$.

The keywords **from** and **to** introduce lower and upper limits on arbitrary things: $\lim\limits_{n\to\infty} \sum\limits_0^n x_i$ 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. Legal characters after **left** and **right** are braces, brackets, bars, **c** and **f** for ceiling and floor, and "" for nothing at all (useful for a right-side-only bracket).

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.

Matrices are made with **matrix**: *matrix { lcol { x sub i above y sub 2 } ccol { 1 above 2 } }* produces $\begin{matrix} x_i & 1 \\ y_2 & 2 \end{matrix}$. In addition, there is **rcol** for a right-justified column.

Diacritical marks are made with **dot**, **dotdot**, **hat**, **tilde**, **bar**, **vec**, **dyad**, and **under**: *x dot = f(t) bar* is $\dot{x}=\overline{f(t)}$, *y dotdot bar ˜=˜ n under* is $\bar{\ddot{y}} = \underline{n}$, and *x vec ˜=˜ y dyad* is $\vec{x} = \overleftrightarrow{y}$.

Sizes and font can be changed with **size** *n* or **size** ±*n*, **roman**, **italic**, **bold**, and **font** *n*. Size and fonts can be changed globally in a document by **gsize** *n* and **gfont** *n*, or by the command-line arguments **−s***n* and **−f***n*.

Normally subscripts and superscripts are reduced by 3 point sizes from the previous size; this may be changed by the command-line argument **−p***n*.

Successive display arguments can be lined up. Place **mark** before the desired lineup point in the first equation; place **lineup** at the place that is to line up vertically in subsequent equations.

Shorthands may be defined or existing keywords redefined with **define**: *define thing % replacement %* defines a new token called *thing* which will be replaced by *replacement* whenever it appears thereafter. The % may be any character that does not occur in *replacement*.

Keywords like *sum* ($\sum$) *int* ($\int$) *inf* (∞) and shorthands like >= ($\geqslant$) −> ($\rightarrow$), and != ($\neq$) are recognized. Greek letters are spelled out in the desired case, as in *alpha* or *GAMMA*. Mathematical words like sin, cos, log are made Roman automatically. *Troff*(1) four-character escapes like \(bs (🅢) can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched; this permits keywords to be entered as text, and can be used to communicate with *troff* when all else fails.

SEE ALSO

troff(1), tbl(1), ms(7), eqnchar(7)
B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics—User's Guide*
J. F. Ossanna, *NROFF/TROFF User's Manual*

BUGS

To embolden digits, parens, etc., it is necessary to quote them, as in 'bold "12.3"'.

NAME

error − analyze and disperse compiler error messages

SYNOPSIS

error [**−n**] [**−s**] [**−q**] [**−v**] [**−t** suffixlist] [**−I** ignorefile] [name]

DESCRIPTION

Error analyzes and optionally disperses the diagnostic error messages produced by a number of compilers and language processors to the source file and line where the errors occurred. It can replace the painful, traditional methods of scribbling abbreviations of errors on paper, and permits error messages and source code to be viewed simultaneously without machinations of multiple windows in a screen editor.

Error looks at the error messages, either from the specified file *name* or from the standard input, and attempts to determine which language processor produced each error message, determines the source file and line number to which the error message refers, determines if the error message is to be ignored or not, and inserts the (possibly slightly modified) error message into the source file as a comment on the line preceeding to which the line the error message refers. Error messages which can't be categorized by language processor or content are not inserted into any file, but are sent to the standard output. *Error* touches source files only after all input has been read. By specifying the **−q** query option, the user is asked to confirm any potentially dangerous (such as touching a file) or verbose action. Otherwise *error* proceeds on its merry business. If the **−t** touch option and associated suffix list is given, *error* will restrict itself to touch only those files with suffices in the suffix list. Error also can be asked (by specifying **−v**) to invoke *vi*(1) on the files in which error messages were inserted; this obviates the need to remember the names of the files with errors.

Error is intended to be run with its standard input connected via a pipe to the error message source. Some language processors put error messages on their standard error file; others put their messages on the standard output. Hence, both error sources should be piped together into *error*. For example, when using the *csh* syntax,

```
make −s lint |& error −q −v
```

will analyze all the error messages produced by whatever programs *make* runs when making lint.

Error knows about the error messages produced by: *make, cc, cpp, ccom, as, ld, lint, pi, pc* and *f77*. *Error* knows a standard format for error messages produced by the language processors, so is sensitive to changes in these formats. For all languages except *Pascal,* error messages are restricted to be on one line. Some error messages refer to more than one line in more than one files; *error* will duplicate the error message and insert it at all of the places referenced.

Error will do one of six things with error messages.

synchronize
: Some language processors produce short errors describing which file it is processing. *Error* uses these to determine the file name for languages that don't include the file name in each error message. These synchronization messages are consumed entirely by *error*.

discard
: Error messages from *lint* that refer to one of the two *lint* libraries, */usr/lib/llib-lc* and */usr/lib/llib-port* are discarded, to prevent accidently touching these libraries. Again, these error messages are consumed entirely by *error.*

nullify
: Error messages from *lint* can be nullified if they refer to a specific function, which is known to generate diagnostics which are not interesting. Nullified error messages are not inserted into the source file, but are written to the standard output. The names of functions to ignore are taken from either the file named *.errorrc* in the

users's home directory, or from the file named by the **−I** option. If the file does not exist, no error messages are nullified. If the file does exist, there must be one function name per line.

not file specific
Error messages that can't be intuited are grouped together, and written to the standard output before any files are touched. They will not be inserted into any source file.

file specific Error message that refer to a specific file, but to no specific line, are written to the standard output when that file is touched.

true errors Error messages that can be intuited are candidates for insertion into the file to which they refer.

Only true error messages are candidates for inserting into the file they refer to. Other error messages are consumed entirely by *error* or are written to the standard output. *Error* inserts the error messages into the source file on the line preceeding the line the language processor found in error. Each error message is turned into a one line comment for the language, and is internally flagged with the string "###" at the beginning of the error, and "%%%" at the end of the error. This makes pattern searching for errors easier with an editor, and allows the messages to be easily removed. In addition, each error message contains the source line number for the line the message refers to. A reasonably formatted source program can be recompiled with the error messages still in it, without having the error messages themselves cause future errors. For poorly formatted source programs in free format languages, such as C or Pascal, it is possible to insert a comment into another comment, which can wreak havoc with a future compilation. To avoid this, format the source program so there are no language statements on the same line as the end of a comment.

Options available with *error* are:

−n Do *not* touch any files; all error messages are sent to the standard output.

−q The user is *queried* whether s/he wants to touch the file. A "y" or "n" to the question is necessary to continue. Absence of the **−q** option implies that all referenced files (except those refering to discarded error messages) are to be touched.

−v After all files have been touched, overlay the visual editor *vi* with it set up to edit all files touched, and positioned in the first touched file at the first error. If *vi* can't be found, try *ex* or *ed* from standard places.

−t Take the following argument as a suffix list. Files whose suffices do not appear in the suffix list are not touched. The suffix list is dot seperated, and "*" wildcards work. Thus the suffix list:

".c.y.foo*.h"

allows *error* to touch files ending with ".c", ".y", ".foo*" and ".y".

−s Print out *statistics* regarding the error categorization. Not too useful.

Error catches interrupt and terminate signals, and if in the insertion phase, will orderly terminate what it is doing.

AUTHOR

Robert Henry

FILES

~/.errorrc	function names to ignore for *lint* error messages
/dev/tty	user's teletype

BUGS

Opens the teletype directly to do user querying.

Source files with links make a new copy of the file with only one link to it.

Changing a language processor's format of error messages may cause *error* to not understand the error message.

Error, since it is purely mechanical, will not filter out subsequent errors caused by 'floodgating' initiated by one syntactically trivial error. Humans are still much better at discarding these related errors.

Pascal error messages belong after the lines affected (error puts them before). The alignment of the '|' marking the point of error is also disturbed by *error.*

Error was designed for work on CRT's at reasonably high speed. It is less pleasant on slow speed terminals, and has never been used on hardcopy terminals.

NAME

ex, edit — text editor

SYNOPSIS

ex [**—**] [**—v**] [**—t** tag] [**—r**] [+*command*] [**—l**] name ...
edit [ex options]

DESCRIPTION

Ex is the root of a family of editors: *edit*, *ex* and *vi*. *Ex* is a superset of *ed*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have not used *ed*, or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed*.

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(1), which is a command which focuses on the display editing portion of *ex*.

DOCUMENTATION

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* 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/ex?.?strings	error messages
/usr/lib/ex?.?recover	recover command
/usr/lib/ex?.?preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup file
/tmp/Ex*nnnnn*	editor temporary
/tmp/Rx*nnnnn*	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), grep(1), vi(1), termcap(5), environ(5)

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

expand, unexpand — expand tabs to spaces, and vice versa

SYNOPSIS

expand [−tabstop] [−tab1,tab2,...,tabn] [file ...]
unexpand [**−a**] [file ...]

DESCRIPTION

Expand processes the named files or the standard input writing the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. *Expand* is useful for pre-processing character files (before sorting, looking at specific columns, etc.) that contain tabs.

If a single *tabstop* argument is given then tabs are set *tabstop* spaces apart instead of the default 8. If multiple tabstops are given then the tabs are set at those specific columns.

Unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default only leading blanks and tabs are reconverted to maximal strings of tabs. If the **−a** option is given, then tabs are inserted whenever they would compress the resultant file by replacing two or more characters.

BUGS

NAME

eyacc — modified yacc allowing much improved error recovery

SYNOPSIS

eyacc [**−v**] [grammar]

DESCRIPTION

Eyacc is an old version of *yacc*(1), which produces tables used by the Pascal system and its error recovery routines. *Eyacc* fully enumerates test actions in its parser when an error token is in the look-ahead set. This prevents the parser from making undesirable reductions when an error occurs before the error is detected. The table format is different in *eyacc* than it was in the old *yacc*, as minor changes had been made for efficiency reasons.

SEE ALSO

yacc(1)
"Practical LR Error Recovery" by Susan L. Graham, Charles B. Haley and W. N. Joy; SIGPLAN Conference on Compiler Construction, August 1979.

AUTHOR

S. C. Johnson

Eyacc modifications by Charles Haley and William Joy.

BUGS

Pc and its error recovery routines should be made into a library of routines for the new *yacc*.

NAME

f77 − Fortran 77 compiler

SYNOPSIS

f77 [option] ... file ...

DESCRIPTION

F77 is the UNIX Fortran 77 compiler. It accepts several types of arguments:

Arguments whose names end with '.f' are taken to be Fortran 77 source programs; they are compiled, and each object program is left on the file in the current directory whose name is that of the source with '.o' substituted for '.f'.

Arguments whose names end with '.r' or '.e' are taken to be Ratfor or EFL source programs, respectively; these are first transformed by the appropriate preprocessor, then compiled by f77.

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

The following options have the same meaning as in *cc*(1). See *ld*(1) for load-time options.

−c Suppress loading and produce '.o' files for each source file.

−g Have the compiler produce additional symbol table information for *sdb*(1). Also pass the **−lg** flag to *ld*(1).

−w Suppress all warning messages. If the option is '−w66', only Fortran 66 compatibility warnings are suppressed.

−p Prepare object files for profiling, see *prof*(1).

Name the final output file *output* instead of 'a.out'.

The following options are peculiar to *f77*.

−onetrip
Compile DO loops that are performed at least once if reached. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)

−u Make the default type of a variable 'undefined' rather than using the default Fortran rules.

−C Compile code to check that subscripts are within declared array bounds.

−F Apply EFL and Ratfor preprocessor to relevant files, put the result in the file with the suffix changed to '.f', but do not compile.

−m Apply the M4 preprocessor to each '.r' or '.e' file before transforming it with the Ratfor or EFL preprocessor.

−E*x* Use the string *x* as an EFL option in processing '.e' files.

−R*x* Use the string *x* as a Ratfor option in processing '.r' files.

Other arguments are taken to be either loader option arguments, or F77-compatible object programs, typically produced by an earlier run, or perhaps libraries of F77-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name 'a.out'.

FILES

file.[fresc]	input file
file.o	object file
a.out	loaded output
/usr/lib/f77pass1	compiler
/lib/f1	pass 2
/lib/c2	optional optimizer

/usr/lib/libF77.a	intrinsic function library
/usr/lib/libI77.a	Fortran I/O library
/lib/libc.a	C library, see section 3

SEE ALSO

S. I. Feldman, P. J. Weinberger, *A Portable Fortran 77 Compiler*
prof(1), cc(1), ld(1), efl(1), ratfor(1)

DIAGNOSTICS

The diagnostics produced by *f77* itself are intended to be self-explanatory. Occasional messages may be produced by the loader.

BUGS

The Fortran 66 subset of the language has been exercised extensively; the newer features have not.

NAME

file – determine file type

SYNOPSIS

file file ...

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ascii, *file* examines the first 512 bytes and tries to guess its language.

BUGS

It often makes mistakes. In particular it often suggests that command files are C programs.

NAME

find − find files

SYNOPSIS

find pathname-list expression

DESCRIPTION

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

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

−perm onum
: True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, see *stat*(2)) become significant and the flags are compared: *(flags&onum)==onum*.

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

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

−user uname
: True if the file belongs to the user *uname* (login name or numeric user ID).

−group gname
: True if the file belongs to group *gname* (group name or numeric group ID).

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

−inum n
: True if the file has inode number *n*.

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

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

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

−ok command
: Like −**exec** except that the generated command is written on the standard output, then the standard input is read and the command executed only upon response **y**.

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

−newer file
: True if the current file has been modified more recently than the argument *file*.

The primaries may be combined using the following operators (in order of decreasing precedence):

1) A parenthesized group of primaries and operators (parentheses are special to the Shell and must be escaped).
2) The negation of a primary ('!' is the unary *not* operator).
3) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).

4) Alternation of primaries ('**−o**' is the *or* operator).

EXAMPLE

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

```
find / \( −name a.out −o −name '*.o' \) −atime +7 −exec rm {} \;
```

FILES

/etc/passwd
/etc/group

SEE ALSO

sh(1), test(1), filsys(5)

BUGS

The syntax is painful.

NAME

finger — user information lookup program

SYNOPSIS

finger [options] name ...

DESCRIPTION

By default *finger* lists the login name, full name, terminal name and write status (as a '*' before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current UNIX user. (Idle time is minutes if it is a single integer, hours and minutes if a ':' is present, or days and hours if a 'd' is present.)

A longer format also exists and is used by *finger* whenever a list of peoples names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file *.plan* in their home directory, and the project on which they are working from the file *.project* also in the home directory.

Finger options include:

−m Match arguments only on user name.

−l Force long output format.

−p Suppress printing of the *.plan* files

−s Force short output format.

FILES

/etc/utmp	who file
/etc/passwd	for users names, offices, ...
/usr/adm/lastlog	last login times
~/.plan	plans
~/.project	projects

SEE ALSO

w(1), who(1)

AUTHOR

Earl T. Cohen

BUGS

Only the first line of the *.project* file is printed.

The encoding of the gcos field is UCB dependent — it knows that an office '197MC' is '197M Cory Hall', and tht '529BE' is '529B Evans Hall'.

A user information data base is in the works and will radically alter the way the information that *finger* uses is stored. Finger will require extensive modification when this is implemented.

NAME

fmt — simple text formatter

SYNOPSIS

fmt [name ...]

DESCRIPTION

Fmt is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces on standard output a version of its input with lines as close to 72 characters long as possible. The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing.

Fmt is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the *ex* editor (e.g. *vi*) the command

!}fmt

will reformat a paragraph, evening the lines.

SEE ALSO

nroff(1), mail(1)

AUTHOR

Kurt Shoens

BUGS

The program was designed to be simple and fast — for more complex operations, the standard text processors are likely to be more appropriate.

NAME

fold — fold long lines for finite width output device

SYNOPSIS

fold [−width] [file ...]

DESCRIPTION

Fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using *expand*(1) before coming to *fold.*

SEE ALSO

expand(1)

BUGS

If underlining is present it may be messed up by folding.

NAME

from − who is my mail from?

SYNOPSIS

from [**−s** sender] [user]

DESCRIPTION

From prints out the mail header lines in your mailbox file to show you who your mail is from. If *user* is specified, then *user*'s mailbox is examined instead of your own. If the -s option is given, then only headers for mail sent by *sender* are printed.

FILES

/usr/spool/mail/*

SEE ALSO

biff(1), mail(1), prmail(1)

NAME

gets − get a string from standard input

SYNOPSIS

gets [default]

DESCRIPTION

N.B.: This command was introduced for use in *.login* scripts when the facilities of the *tset*(1) command were not totally adequate in setting the terminal type. This is no longer true, and *gets* should no longer be needed. To boot, a construct "$<" is available in *csh*(1) now which has the functionality of *gets:*

```
set a=$<
if ($a == '') set a=default
```

replaces

```
set a=`gets default`
```

Users of *sh*(1) should use its *read* command rather than *gets.*

Gets can be used with *csh*(1) to read a string from the standard input. If a *default* is given it is used if just return is typed, or if an error occurs. The resultant string (either the default or as read from the standard input is written to the standard output. If no *default* is given and an error occurs, *gets* exits with exit status 1.

SEE ALSO

csh (1)

BUGS

Gets is obsolete.

NAME

graph − draw a graph

SYNOPSIS

graph [option] ...

DESCRIPTION

Graph with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. Successive points are connected by straight lines. The graph is encoded on the standard output for display by the *plot*(1) filters.

If the coordinates of a point are followed by a nonnumeric string, that string is printed as a label beginning on the point. Labels may be surrounded with quotes "...", in which case they may be empty or contain blanks and numbers; labels never contain newlines.

The following options are recognized, each as a separate argument.

−a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument (default 1). A second optional argument is the starting point for automatic abscissas (default 0 or lower limit given by **−x**).

−b Break (disconnect) the graph after each label in the input.

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

−g Next argument is grid style, 0 no grid, 1 frame with ticks, 2 full grid (default).

−l Next argument is label for graph.

−m Next argument is mode (style) of connecting lines: 0 disconnected, 1 connected (default). Some devices give distinguishable line styles for other small integers.

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

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

−y [l]
Similarly for *y*.

−h Next argument is fraction of space for height.

−w Similarly for width.

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

−u Similarly to move up before plotting.

−t Transpose horizontal and vertical axes. (Option **−x** now applies to the vertical axis.)

A legend indicating grid range is produced with a grid unless the **−s** option is present.

If a specified lower limit exceeds the upper limit, the axis is reversed.

SEE ALSO

spline(1), plot(1)

BUGS

Graph stores all points internally and drops those for which there isn't room.
Segments that run out of bounds are dropped, not windowed.
Logarithmic axes may not be reversed.

NAME

grep, egrep, fgrep — search a file for a pattern

SYNOPSIS

grep [option] ... expression [file] ...

egrep [option] ... [expression] [file] ...

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. *Grep* patterns are limited regular expressions in the style of *ex*(1); it uses a compact nondeterministic algorithm. *Egrep* patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. *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 (*fgrep* only).

−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.

−i The case of letters is ignored in making comparisons. (E.g. upper and lower case are considered identical.) (*grep* and *fgrep* only)

−s Silent mode. Nothing is printed (except error messages). This is useful for checking the error status.

−w The expression is searched for as a word (as if surrounded by '\<' and '\>', see *ex*(1).) (*grep* only)

−e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a −.

−f *file* The regular expression (*egrep*) or 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*.

Egrep accepts extended regular expressions. In the following description 'character' excludes newline:

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.

SEE ALSO

ex(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 of a stream

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.

SEE ALSO
tail(1)

NAME

iostat — report I/O statistics

SYNOPSIS

iostat [interval [count]]

DESCRIPTION

Iostat iteratively reports the number of characters read and written to terminals, and, for each disk, the number of seeks and transfers per second, and the milliseconds per average seek. It also gives the percentage of time the system has spent in user mode, in user mode running low priority (niced) processes, in system mode, and idling.

To compute this information, for each disk, seeks and data transfer completions and number of words transferred are counted; for terminals collectively, the number of input and output characters are counted. Also, each sixtieth of a second, the state of each disk is examined and a tally is made if the disk is active. From these numbers and given the transfer rates of the devices it is possible to determine average seek times for each device.

The optional *interval* argument causes *iostat* to report once each *interval* seconds. The first report is for all time since a reboot and each subsequent report is for the last interval only.

The optional *count* argument restricts the number of reports.

FILES

/dev/kmem /vmunix

SEE ALSO

vmstat(1)

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, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

Fields are normally separated by blank, tab or newline. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

−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.

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 ...
kill −l

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 *signal*(2)). The signal names are listed by 'kill −l', and are as given in */usr/include/signal.h*, stripped of the common SIG prefix.

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

To shut the system down and bring it up single user the super-user may send the initialization process a TERM (terminate) signal by 'kill 1'; see *init*(8). To force *init* to close and open terminals according to what is currently in /etc/ttys use 'kill −HUP 1' (sending a hangup, signal 1).

The process number of an asynchronous process started with '&' is reported by the shell. Process numbers can also be found by using *Kill* is a built-in to *csh*(1); it allows job specifiers "%..." so process id's are not as often used as *kill* arguments. See *csh*(1) for details.

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 should be provided.

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 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* will print the sessions of the specified users and teletypes, most recent first, indicating the times at which the session began, the duration of the session, and the teletype which the session took place on. If the session is still continuing or was cut short by a reboot, *last* so indicates.

The pseudo-user **reboot** logs in at reboots of the system, thus

last reboot

will give an indication of mean time between reboot.

Last with no arguments prints a record of all logins and logouts, in reverse order.

If *last* is interrupted, it indicates how far the search has progressed in *wtmp*. If interrupted with a quit signal (generated by a control-\) *last* indicates how far the search has progressed so far, and the search continues.

FILES

/usr/adm/wtmp login data base
/usr/adm/shutdownlog which records shutdowns and reasons for same

SEE ALSO

wtmp(5), ac(8), lastcomm(1)

AUTHOR

Howard Katseff

NAME

lastcomm − show last commands executed in reverse order

SYNOPSIS

lastcomm [command name] ... [user name] ...

DESCRIPTION

Lastcomm gives information on previously executed commands. *Lastcomm* with no arguments prints information about all the commands recorded during the current accounting file's lifetime. If called with arguments, only those accounting entries whose command name or user name matches one of the arguments are printed. So, for example,

lastcomm a.out

would produce a listing of all the executions of commands named a.out, and

lastcomm root

would produce a listing of all the commands executed by user root.

For each process entry, the following are printed.

The name of the user who ran the process.
The command name under which the process was called.
The amount of cpu time used by the process (in seconds).
The time the process exited.

AUTHOR

Len Edmondson

SEE ALSO

last (1)

NAME

ld − link editor

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 (unless the **−e** option is specified).

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by *ranlib*(1), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. The first member of a library should be a file named '__.SYMDEF', which is understood to be a dictionary for the library as produced by *ranlib*(1); the dictionary is searched iteratively to satisfy as many references as possible.

The symbols '_etext', '_edata' and '_end' ('etext', 'edata' and 'end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. It is erroneous to define these symbols.

Ld understands several options. Except for **−l**, they should appear before the file names.

−A This option specifies incremental loading, i.e. linking is to be done in a manner so that the resulting object may be read into an already executing program. The next argument is the name of a file whose symbol table will be taken as a basis on which to define additional symbols. Only newly linked material will be entered into the text and data portions of **a.out**, but the new symbol table will reflect every symbol defined before and after the incremental load. This argument must appear before any other object file in the argument list. The **−T** option may be used as well, and will be taken to mean that the newly linked segment will commence at the corresponding address (which must be a multiple of 1024). The default value is the old value of _end.

−D Take the next argument as a hexadecimal number and pad the data segment with zero bytes to the indicated length.

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

−e The following argument is taken to be the name of the entry point of the loaded program; location 0 is the default.

−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.

−M produce a primitive load map, listing the names of the files which will be loaded.

−N Do not make the text portion read only or sharable. (Use "magic number" 0407.)

−n Arrange (by giving the output file a 0410 "magic number") 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 1024 byte boundary following the end of the text.

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

−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.

−S 'Strip' the output by removing all symbols except locals and globals.

−s 'Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debuggers). This information can also be removed by *strip*(1).

−T The next argument is a hexadecimal number which sets the text segment origin. The default origin is 0.

−t ("trace") Print the name of each file as it is processed.

−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.

−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.

−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.

−y*sym* Indicate each file in which *sym* appears, its type and whether the file defines or references it. Many such options may be given to trace many symbols. (It is usually necessary to begin *sym* with an '_', as external C, FORTRAN and Pascal variables begin with underscores.)

−z Arrange for the process to be loaded on demand from the resulting executable file (413 format) rather than preloaded. This is the default. Results in a 1024 byte header on the output file followed by a text and data segment each of which have size a multiple of 1024 bytes (being padded out with nulls in the file if necessary). With this format the first few BSS segment symbols may actually appear (from the output of *size*(2)) to live in the data segment; this to avoid wasting the space resulting from data segment size roundup.

FILES

/lib/lib*.a	libraries
/usr/lib/lib*.a	more libraries
/usr/local/lib/lib*.a	still more libraries
a.out	output file

SEE ALSO

as(1), ar(1), cc(1), ranlib(1)

BUGS

There is no way to force data to be page aligned.

NAME

learn — computer aided instruction about UNIX

SYNOPSIS

learn [−directory] [subject [lesson [speed]]]

DESCRIPTION

Learn gives CAI courses and practice in the use of UNIX. To get started simply type 'learn'. The program will ask questions to find out what you want to do. The questions may be bypassed by naming a *subject*, and the last *lesson* number that *learn* told you in the previous session. You may also include a *speed* number that was given with the lesson number (but without the parentheses that *learn* places around the speed number). If *lesson* is '−', *learn* prompts for each lesson; this is useful for debugging.

The *subjects* presently handled are

editor
eqn
files
macros
morefiles
C

The special command 'bye' terminates a *learn* session.

The −*directory* option allows one to exercise a script in a nonstandard place.

FILES

/usr/learn and all dependent directories and files

BUGS

The main strength of *learn*, that it asks the student to use the real UNIX, also makes possible baffling mistakes. It is helpful, especially for nonprogrammers, to have a UNIX initiate near at hand during the first sessions.

Occasionally lessons are incorrect, sometimes because the local version of a command operates in a non-standard way. Such lessons may be skipped, but it takes some sophistication to recognize the situation.

NAME

leave − remind you when you have to leave

SYNOPSIS

leave [hhmm]

DESCRIPTION

Leave waits until the specified time, then reminds you that you have to leave. You are reminded 5 minutes and 1 minute before the actual time, at the time, and every minute thereafter. When you log off, *leave* exits just before it would have printed the next message.

The time of day is in the form hhmm where hh is a time in hours (on a 12 or 24 hour clock). All times are converted to a 12 hour clock, and assumed to be in the next 12 hours.

If no argument is given, *leave* prompts with "When do you have to leave?". A reply of newline causes *leave* to exit, otherwise the reply is assumed to be a time. This form is suitable for inclusion in a *.login* or *.profile.*

Leave ignores interrupts, quits, and terminates. To get rid of it you should either log off or use "kill -9" giving its process id.

SEE ALSO

calendar(1)

AUTHOR

Mark Horton

BUGS

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 following *lex* program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%%
[A−Z] putchar(yytext[0]+'a'−'A');
[ ]+$
[ ]+    putchar(' ');
```

The options have the following meanings.

−t Place the result on the standard output instead of in file 'lex.yy.c'.

−v Print a one-line summary of statistics of the generated analyzer.

−n Opposite of **−v**; **−n** is default.

−f 'Faster' compilation: don't bother to pack the resulting tables; limited to small programs.

SEE ALSO

yacc(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.

FILES

/usr/lib/lint/lint[12] programs
/usr/lib/lint/llib-lc declarations for standard functions
/usr/lib/lint/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

lisp — lisp interpreter

SYNOPSIS

lisp

DESCRIPTION

Lisp is a lisp interpreter for a dialect which closely resembles MIT's MACLISP. This lisp, known as FRANZ LISP, features an I/O facility which allows the user to change the input and output syntax, add macro characters, and maintain compatibility with upper-case only lisp systems; infinite precision integer arithmetic, and an error facility which allows the user to trap system errors in many different ways. Interpreted functions may be mixed with code compiled by *liszt*(1) and both may be debugged using the "Joseph Lister" trace package. A *lisp* containing compiled and interpreted code may be dumped into a file for later use.

There are too many functions to list here; one should refer to the manuals listed below.

AUTHORS

An early version was written by Jeff Levinsky, Mike Curry, and John Breedlove. Keith Sklower wrote and is maintaining the current version, with the assistance of John Foderaro. The garbage collector was implemented by Bill Rowan.

FILES

/usr/lib/lisp/auxfns0.l	common functions
/usr/lib/lisp/auxfns1.l	less common functions
/usr/lib/lisp/trace.l	Joseph Lister trace package

SEE ALSO

liszt(1)
'FRANZ LISP Manual, Version 1' by John K. Foderaro
MACLISP Manual

BUGS

The error system is in a state of flux and not all error messages are as informative as they could be.

NAME

liszt – compile a Franz Lisp program

SYNOPSIS

liszt [−mpqruwxCQST] [−o objfile] [name]

DESCRIPTION

Liszt takes a file whose names ends in '.l' and compiles the FRANZ LISP code there leaving an object program on the file whose name is that of the source with '.o' substituted for '.l'.

The following options are interpreted by *liszt.*

−m Compile a MACLISP file, by changing the readtable to conform to MACLISP syntax and including a macro-defined compatibility package.

−o Put the object code in the specified file, rather than the default '.o' file.

−p places profiling code at the beginning of each non-local function. If the lisp system is also created with profiling in it, this allows function calling frequency to be determined (see *prof*(1).)

−q Only print warning and error messages. Compilation statistics and notes on correct but unusual constructs will not be printed.

−r place bootstrap code at the beginning of the object file, which when the object file is executed will cause a lisp system to be invoked and the object file fasl'ed in.

−u Compile a UCI-lispfile, by changing the readtable to conform to UCI-Lisp syntax and including a macro-defined compatibility package.

−w Suppress warning diagnostics.

−x Create a lisp cross reference file with the same name as the source file but with '.x' appended. The program *lxref*(1) reads this file and creates a human readable cross reference listing.

−C put comments in the assembler output of the compiler. Useful for debugging the compiler.

-Q Print compilation statistics and warn of strange constructs. This is the default.

−S Compile the named program and leave the assembler-language output on the corresponding file suffixed '.s'. This will also prevent the assembler language file from being assembled.

−T send the assembler output to standard output.

If no source file is specified, then the compiler will run interactively. You will find yourself talking to the *lisp*(1) top-level command interpreter. You can compile a file by using the function *liszt* (an nlambda) with the same arguments as you use on the command line. For example to compile 'foo', a MACLISP file, you would use:

(liszt −m foo)

Note that *liszt* supplies the ".l" extension for you.

FILES

/usr/lib/lisp/machacks.l	MACLISP compatibility package
/usr/lib/lisp/syscall.l	macro definitions of Unix system calls
/usr/lib/lisp/ucifnc.l	UCI Lisp compatibility package

AUTHOR

John Foderaro

SEE ALSO

lisp(1), lxref(1)

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. 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 to a directory or to link across file systems.

SEE ALSO

rm(1), cp(1), mv(1)

NAME

lock — reserve a terminal

SYNOPSIS

lock

DESCRIPTION

Lock requests a password from the user, then prints "LOCKED" on the terminal and refuses to relinquish the terminal until the password is repeated. If the user forgets the password, he has no other recourse but to login elsewhere and kill the lock process.

AUTHOR

Kurt Shoens

BUGS

Should timeout after 15 minutes.

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 latter case is the one summarized above and described here. See "How to Get Started" for how to dial up initially.

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

After a successful login, accounting files are updated and the user is informed of the existence of mail, and the message of the day is printed, as is the time he last logged in (unless he has a ".hushlogin" file in his home directory − this is mostly used to make life easier for non-human users, such as *uucp*).

Login initializes the user and group IDs and the working directory, then executes a command interpreter (usually *sh*(1)) according to specifications found in a password file. Argument 0 of the command interpreter is "−sh", or more generally the name of the command interpreter with a leading dash ("−") prepended.

Login also initializes the environment *environ*(5) with information specifying home directory, command interpreter, terminal type (if available) and user name.

If the file /etc/nologin exists *login* prints its contents on the user's terminal and exits. This is used by *shutdown*(8) to stop users logging in when the system is about to go down.

Login is recognized by *sh*(1) and *csh*(1) and executed directly (without forking).

FILES

/etc/utmp	accounting
/usr/adm/wtmp	accounting
/usr/spool/mail/*	mail
/etc/motd	message-of-the-day
/etc/passwd	password file
/etc/nologin	stops logins
.hushlogin	makes login quieter
/etc/securetty	lists ttys that root may log in on

SEE ALSO

init(8), newgrp(1), getty(8), mail(1), passwd(1), passwd(5), environ(5), shutdown(8)

DIAGNOSTICS

"Login incorrect," if the name or the password is bad.
"No Shell", "cannot open password file", "no directory": consult a programming counselor.

NAME

lorder — find ordering relation for an object library

SYNOPSIS

lorder file ...

DESCRIPTION

The input is one or more object or library archive (see *ar*(1)) *files*. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(1) to find an ordering of a library suitable for one-pass access by *ld*(1).

This brash one-liner intends to build a new library from existing '.o' files.

```
ar cr library `lorder *.o | tsort`
```

The need for lorder may be vitiated by use of *ranlib*(1), which converts an ordered archive into a randomly accessed library.

FILES

*symref, *symdef
nm(1), sed(1), sort(1), join(1)

SEE ALSO

tsort(1), ld(1), ar(1), ranlib(1)

BUGS

The names of object files, in and out of libraries, must end with '.o'; nonsense results otherwise.

NAME

lpr, lprm, lpq, print — line printer spooler

SYNOPSIS

lpr [**−m**] [name ...]
lprm [id ...] [filename ...] [owner ...]
lpq
print file ...,

DESCRIPTION

Lpr causes the named files to be queued for printing. If no files are named, the standard input is read. The option **−m** causes notification via *mail*(1) to be sent when the job completes.

Lprm removes an entry from the line printer queue. The id, filename or owner should be that reported by *lpq*. All appropriate files will be removed. The id of each file removed from the queue will be printed.

Lpq prints the line printer queue. Each entry in the queue is printed showing the owner of the queue entry, an identification number, the size of the entry in characters, and the file which is to be printed. The *id* is useful for removing a specific entry from the printer queue using *lprm*(1).

Print pr's a copy of each named file on the line printer. It is a one line shell script:

```
pr $* | lpr
```

FILES

/usr/spool/lpd/*	spool area
/usr/lib/lpd	printer daemon
/usr/lib/lpf	filter to handle banners and underlining

SEE ALSO

pr(1)

BUGS

NAME

ls — list contents of directory

SYNOPSIS

ls [**−abcdfgilmqrstux1CFR**] name ...
l [*ls* options] name ...

DESCRIPTION

For each directory argument, *ls* lists the contents of the directory; for each file argument, *ls* repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The format chosen depends on whether the output is going to a teletype, and may also be controlled by option flags. The default format for a teletype is to list the contents of directories in multi-column format, with the entries sorted down the columns. (Files which are not the contents of a directory being interpreted are always sorted across the page rather than down the page in columns. This is because the individual file names may be arbitrarily long.) If the standard output is not a teletype, the default format is to list one entry per line. Finally, there is a stream output format in which files are listed across the page, separated by ',' characters. The **−m** flag enables this format; when invoked as *l* this format is also used.

There are an unbelievable number of options:

−l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

−t Sort by time modified (latest first) instead of by name, as is normal.

−a List all entries; usually '.' and '..' are suppressed.

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

−d If argument is a directory, list only its name, not its contents (mostly used with **−l** to get status on directory).

−r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

−u Use time of last access instead of last modification for sorting (**−t**) or printing (**−l**).

−c Use time of file creation for sorting or printing.

−i Print i-number in first column of the report for each file listed.

−f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **−l**, **−t**, **−s**, and **−r**, and turns on **−a**; the order is the order in which entries appear in the directory.

−g Give group ID instead of owner ID in long listing.

−m force stream output format

−1 force one entry per line output format, e.g. to a teletype

−C force multi-column output, e.g. to a file or a pipe

−q force printing of non-graphic characters in file names as the character '?'; this normally happens only if the output device is a teletype

−b force printing of non-graphic characters to be in the \ddd notation, in octal.

−x force columnar printing to be sorted across rather than down the page; this is the default if the last character of the name the program is invoked with is an 'x'.

−F cause directories to be marked with a trailing '/' and executable files to be marked with a trailing '*'; this is the default if the last character of the name the program is invoked with is a 'f'.

−R recursively list subdirectories encountered.

The mode printed under the **−l** option contains 11 characters which are interpreted as follows: the first character is

d if the entry is a directory;
b if the entry is a block-type special file;
c if the entry is a character-type special file;
m if the entry is a multiplexor-type character special file;
− if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable;
− if the indicated permission is not granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise the user-execute permission character is given as **s** if the file has set-user-ID mode.

The last character of the mode (normally 'x' or '−') is **t** if the 1000 bit of the mode is on. See *chmod*(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for 'ls −l'.
/etc/group to get group ID's for 'ls −g'.

BUGS

Newline and tab are considered printing characters in file names.

The output device is assumed to be 80 columns wide.

The option setting based on whether the output is a teletype is undesirable as "ls −s" is much different than "ls −s | lpr". On the other hand, not doing this setting would make old shell scripts which used *ls* almost certain losers.

Column widths choices are poor for terminals which can tab.

NAME

lxref — lisp cross reference program

SYNOPSIS

lxref [−N] file ...

DESCRIPTION

Lxref reads cross reference file(s) written by the lisp compiler *liszt* and prints a cross reference listing on the standard output. *Liszt* will create a cross reference file during compilation when it is given the −x switch. Cross reference files usually end in '.x' and consequently *lxref* will append a '.x' to the file names given if necessary. The one option to *lxref* is a decimal integer, N, which sets the *ignorelevel*. If a function is called more than *ignorelevel* times, the cross reference listing will just print the number of calls instead of listing each one of them. The default for *ignorelevel* is 50.

AUTHOR

John Foderaro

SEE ALSO

lisp(1), liszt(1)

BUGS

NAME

m4 − macro processor

SYNOPSIS

m4 [files]

DESCRIPTION

M4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no arguments, or if an argument is '−', the standard input is read. The processed text is written on the standard output.

Macro calls have the form

    name(arg1,arg2, . . . , argn)

The '(' must immediately follow the name of the macro. If a defined macro name is not followed by a '(', it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Potential macro names consist of alphabetic letters, digits, and underscore '_', where the first character is not a digit.

Left and right single quotes (`') are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define — The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of $*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string.

undefine — removes the definition of the macro named in its argument.

ifdef — If the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word *unix* is predefined on UNIX versions of *m4*.

changequote — Change quote characters to the first and second arguments. *Changequote* without arguments restores the original values (i.e., `').

divert — *M4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert — causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum — returns the value of the current output stream.

dnl — reads and discards characters up to and including the next newline.

ifelse — has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

incr returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

eval evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, −, *, /, %, ^ (exponentiation); relationals; parentheses.

len returns the number of characters in its argument.

index returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.

substr returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include returns the contents of the file named in the argument.

sinclude is identical to *include,* except that it says nothing if the file is inaccessible.

syscmd executes the UNIX command given in the first argument. No value is returned.

maketemp fills in a string of XXXXX in its argument with the current process id.

errprint prints its argument on the diagnostic output file.

dumpdef prints current names and definitions, for the named items, or for all if no arguments are given.

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The M4 Macro Processor*

NAME

mail − send and receive mail

SYNOPSIS

mail [**−f** [name]] [people ...]

INTRODUCTION

Mail is a intelligent mail processing system, which has a command syntax reminiscent of *ed* with lines replaced by messages.

Sending mail. To send a message to one or more other people, *mail* can be invoked with arguments which are the names of people to send to. You are then expected to type in your message, followed by an EOT (control−D) at the beginning of a line. The section below, labeled *Replying to or originating mail,* describes some features of *mail* available to help you compose your letter.

Reading mail. In normal usage, *mail* is given no arguments and checks your mail out of the post office, then printing out a one line header of each message there. The current message is initially the first message (numbered 1) and can be printed using the **print** command (which can be abbreviated **p**). You can move among the messages much as you move between lines in *ed,* with the commands '+' and '−' moving backwards and forwards, and simple numbers typing the addressed message.

Disposing of mail. After examining a message you can **delete** (**d**) the message or **reply** (**r**) to it. Deletion causes the *mail* program to forget about the message. This is not irreversible, the message can be **undeleted** (**u**) by giving its number, or the *mail* session can be aborted by giving the **exit** (**x**) command. Deleted messages will, however, usually disappear never to be seen again.

Specifying messages. Commands such as **print** and **delete** often can be given a list of message numbers as argument to apply to a number of messages at once. Thus "delete 1 2" deletes messages 1 and 2, while "delete 1−5" deletes messages 1 through 5. The special name "*" addresses all messages, and "$" addresses the last message; thus the command **top** which prints the first few lines of a message could be used in "top *" to print the first few lines of all messages.

Replying to or originating mail. You can use the **reply** command to set up a response to a message, sending it back to the person who it was from. Text you then type in, up to an end-of-file (or a line consisting only of a ".") defines the contents of the message. While you are composing a message, *mail* treats lines beginning with the character '~' specially. For instance, typing "~m" (alone on a line) will place a copy of the current message into the response right shifting it by a tabstop. Other escapes will set up subject fields, add and delete recipients to the message and allow you to escape to an editor to revise the message or to a shell to run some commands. (These options will be given in the summary below.)

Ending a mail processing session. You can end a *mail* session with the **quit** (**q**) command. Messages which have been examined go to your *mbox* file unless they have been deleted in which case they are discarded. Unexamined messages go back to the post office. The **−f** option causes *mail* to read in the contents of your *mbox* (or the specified file) for processing; when you **quit** *mail* writes undeleted messages back to this file.

Personal and systemwide distribution lists. It is also possible to create a personal distribution lists so that, for instance, you can send mail to "cohorts" and have it go to a group of people. Such lists can be defined by placing a line like

alias cohorts bill ozalp sklower jkf mark cory:kridle in the file .mailrc in your home directory. The current list of such aliases can be displayed by the **alias (a)** command in *mail*. System wide distribution lists can be created by editing /usr/lib/aliases, see *aliases*(5) and *delivermail*(8); these are kept in a slightly different syntax. In mail you send, personal aliases will be expanded in mail sent to others so that they will be able to **reply** to the recipients. System wide *aliases* are not expanded when the mail is sent, but any reply returned to the machine will have the system wide alias expanded as all mail goes through *delivermail*. If you edit /usr/lib/aliases, you must run the program *newaliases*(1).

Network mail (ARPA, UUCP, Berknet) Mail to sites on the ARPA network and sites within Bell laboratories can be sent using "name@site" for ARPA-net sites or "machine!user" for Bell labs sites, provided appropriate gateways are known to the system. (Be sure to escape the ! in Bell sites when giving it on a *csh* command line by preceding it with an \. Machines on an instance of the Berkeley network are addressed as "machine:user", e.g. "csvax:bill". When addressed from the arpa-net, "csvax:bill" is known as "csvax.bill@berkeley".

Mail has a number of options which can be **set** in the *.mailrc* file to alter its behavior; thus "set askcc" enables the "askcc" feature. (These options are summarized below.)

SUMMARY

(Adapted from the 'Mail Reference Manual') Each command is typed on a line by itself, and may take arguments following the command word. The command need not be typed in its entirety — the first command which matches the typed prefix is used. For the commands which take message lists as arguments, if no message list is given, then the next message forward which satisfies the command's requirements is used. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, *mail* types "No applicable messages" and aborts the command.

— Goes to the previous message and prints it out. If given a numeric argument *n* , goes to the *n th* previous message and prints it.

? Prints a brief summary of commands.

! Executes the UNIX shell command which follows.

alias (**a**) With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, adds the users named in the second and later arguments to the alias named in the first argument.

chdir (**c**) Changes the user's working directory to that specified, if given. If no directory is given, then changes to the user's login directory.

delete (**d**) Takes a list of messages as argument and marks them all as deleted. Deleted messages will not be saved in *mbox* , nor will they be available for most other commands.

dp (also **dt**) Deletes the current message and prints the next message. If there is no next message, *mail* says "at EOF."

edit (**e**) Takes a list of messages and points the text editor at each one in turn. On return from the editor, the message is read back in.

exit (**ex** or **x**) Effects an immediate return to the Shell without modifying the user's system mailbox, his *mbox* file, or his edit file in **−f** .

from (**f**) Takes a list of messages and prints their message headers.

headers (**h**) Lists the current range of headers, which is an 18 message group. If a "+" argument is given, then the next 18 message group is printed, and if a "−" argument is given, the previous 18 message group is printed.

help A synonym for ?

hold (**ho**, also **preserve**) Takes a message list and marks each message therein to be saved in the user's system mailbox instead of in *mbox*. Does not override the **delete** command.

mail (**m**) Takes as argument login names and distribution group names and sends mail to those people.

next (**n** like + or CR) Goes to the next message in sequence and types it. With an argument list, types the next matching message.

preserve A synonym for **hold**.

print (**p**) Takes a message list and types out each message on the user's terminal.

quit (**q**) Terminates the session, saving all undeleted, unsaved messages in the user's *mbox* file in his login directory, preserving all messages marked with **hold** or **preserve** or never referenced in his system mailbox, and removing all other messages from his system mailbox. If new mail has arrived during the session, the message "You have new mail" is given. If given while editing a mailbox file with the **−f** flag, then the edit file is rewritten. A return to the Shell is effected, unless the rewrite of edit file fails, in which case the user can escape with the **exit** command.

reply (**r**) Takes a message list and sends mail to each message author just like the **mail** command. The default message must not be deleted.

respond A synonym for **reply** .

save (**s**) Takes a message list and a filename and appends each message in turn to the end of the file. The filename in quotes, followed by the line count and character count is echoed on the user's terminal.

set (**se**) With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form "option=value" or "option."

shell (**sh**) Invokes an interactive version of the shell.

size Takes a message list and prints out the size in characters of each message.

top Takes a message list and prints the top few lines of each. The number of lines printed is controlled by the variable **toplines** and defaults to five.

type (**t**) A synonym for **print** .

unalias Takes a list of names defined by **alias** commands and discards the remembered groups of users. The group names no longer have any significance.

undelete (**u**) Takes a message list and marks each one as *not* being deleted.

unset Takes a list of option names and discards their remembered values; the inverse of **set** .

visual (**v**) Takes a message list and invokes the display editor on each message.

write (**w**) A synonym for **save** .

xit (**x**) A synonym for **exit** .

Here is a summary of the tilde escapes, which are used when composing messages to perform special functions. Tilde escapes are only recognized at the beginning of lines. The name "tilde escape" is somewhat of a misnomer since the actual escape character can be set by the option **escape**.

~!command Execute the indicated shell command, then return to the message.

~c name ... Add the given names to the list of carbon copy recipients.

~d Read the file "dead.letter" from your home directory into the message.

~e Invoke the text editor on the message collected so far. After the editing session is finished, you may continue appending text to the message.

~h Edit the message header fields by typing each one in turn and allowing the user to append text to the end or modify the field by using the current terminal erase and kill characters.

~m messages
Read the named messages into the message being sent, shifted right one tab. If no messages are specified, read the current message.

~p Print out the message collected so far, prefaced by the message header fields.

~q Abort the message being sent, copying the message to "dead.letter" in your home directory if **save** is set.

~r filename Read the named file into the message.

~s string Cause the named string to become the current subject field.

~t name ... Add the given names to the direct recipient list.

~v Invoke an alternate editor (defined by the VISUAL option) on the message collected so far. Usually, the alternate editor will be a screen editor. After you quit the editor, you may resume appending text to the end of your message.

~w filename Write the message onto the named file.

~|command Pipe the message through the command as a filter. If the command gives no output or terminates abnormally, retain the original text of the message. The command *fmt*(1) is often used as *command* to rejustify the message.

~~string Insert the string of text in the message prefaced by a single ~. If you have changed the escape character, then you should double that character in order to send it.

Options are controlled via the **set** and **unset** commands. Options may be either binary, in which case it is only significant to see whether they are set or not, or string, in which case the actual value is of interest. The binary options include the following:

append Causes messages saved in *mbox* to be appended to the end rather than prepended. (This is set in /usr/lib/Mail.rc on version 7 systems.)

ask Causes *mail* to prompt you for the subject of each message you send. If you respond with simply a newline, no subject field will be sent.

askcc Causes you to be prompted for additional carbon copy recipients at the end of each message. Responding with a newline indicates your satisfaction with the current list.

autoprint Causes the **delete** command to behave like **dp** — thus, after deleting a message, the next one will be typed automatically.

ignore Causes interrupt signals from your terminal to be ignored and echoed as @'s.

metoo Usually, when a group is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.

quiet Suppresses the printing of the version when first invoked.

save Causes the message collected prior to a interrupt to be saved on the file "dead.letter" in your home directory on receipt of two interrupts (or after a **~q**.)

The following options have string values:

EDITOR	Pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then a default editor is used.
SHELL	Pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.
VISUAL	Pathname of the text editor to use in the **visual** command and ~v escape.
escape	If defined, the first character of this option gives the character to use in the place of ~ to denote escapes.
record	If defined, gives the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not so saved.
toplines	If defined, gives the number of lines of a message to be printed out with the **top** command; normally, the first five lines are printed.

FILES

/usr/spool/mail/*	post office
~/mbox	your old mail
~/.mailrc	file giving initial mail commands
/tmp/R#	temporary for editor escape
/usr/lib/Mail.help*	help files
/usr/lib/Mail.rc	system initialization file
/bin/mail	to do actual mailing
/etc/delivermail	postman

SEE ALSO

binmail(1), fmt(1), newaliases(1), aliases(5), delivermail(8)
'The Mail Reference Manual'

AUTHOR

Kurt Shoens

BUGS

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, 'FFLAGS' for *f77*(1) options, 'PFLAGS' for *pc*(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, Makefile

SEE ALSO

sh(1), touch(1), f77(1), pc(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

man — find manual information by keywords; print out the manual

SYNOPSIS

man **−k** keyword ...
man **−f** file ...
man [**−**] [**−t**] [section] title ...

DESCRIPTION

Man is a program which gives information from the programmers manual. It can be asked form one line descriptions of commands specified by name, or for all commands whose description contains any of a set of keywords. It can also provide on-line access to the sections of the printed manual.

When given the option **−k** and a set of keywords, *man* prints out a one line synopsis of each manual sections whose listing in the table of contents contains that keyword.

When given the option **−f** and a list of file names, *man* attempts to locate manual sections related to those files, printing out the table of contents lines for those sections.

When neither **−k** nor **−f** is specified, *man* formats a specified set of manual pages. If a section specifier is given *man* looks in the that section of the manual for the given *titles. Section* is an arabic section number, i.e. 3, which may be followed by a single letter classifier, i.e. 1g indicating a graphics program in section 1. If *section* is omitted, *man* searches all sections of the manual, giving preference to commands over subroutines in system libraries, and printing the first section it finds, if any.

If the standard output is a teletype, or if the flag **−** is given, then *man* pipes its output through *cat*(1) with the option **−s** to crush out useless blank lines, *ul*(1) to create proper underlines for different terminals, and through *more*(1) to stop after each page on the screen. Hit a space to continue, a control-D to scroll 11 more lines when the output stops.

The **−t** flag causes *man* to arrange for the specified section to be *troff'ed* to a suitable raster output device; see *vtroff*(1).

FILES

/usr/man/man?/*
/usr/man/cat?/*

SEE ALSO

more(1), ul(1), whereis(1), catman(8)

BUGS

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

NAME

mesg − permit or deny messages

SYNOPSIS

mesg [n] [y]

DESCRIPTION

Mesg with argument n forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *Mesg* with argument y reinstates permission. All by itself, *mesg* reports the current state without changing it.

FILES

/dev/tty*

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. Standard entries, '.', for the directory itself, and '..' for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

rm(1)

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made. Otherwise it prints a diagnostic and returns nonzero.

NAME

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 null character and a new-line character; the null character terminates the message so it can be easily used when retrieved, the new-line character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file which can be used to retrieve the message, i.e.:

```
char    efilname[] = "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
{
        char buf[256];

        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
oops:
                        perror(efilname);
                        exit(1);
                }
        }
        if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

The optional — causes the error messages to be placed at the end of the specified message file for recompiling part of a large *mkstr* ed program.

SEE ALSO

lseek(2), xstr(1)

AUTHORS

William Joy and Charles Haley

NAME

more, page — file perusal filter for crt viewing

SYNOPSIS

more [**−cdflsu**] [−*n*] [+*linenumber*] [+/*pattern*] [name ...]

page *more options*

DESCRIPTION

More is a filter which allows examination of a continuous text one screenful at a time on a soft-copy terminal. It normally pauses after each screenful, printing --More-- at the bottom of the screen. If the user then types a carriage return, one more line is displayed. If the user hits a space, another screenful is displayed. Other possibilites are enumerated later.

The command line options are:

−*n* An integer which is the size (in lines) of the window which *more* will use instead of the default.

−c *More* will draw each page by beginning at the top of the screen and erasing each line just before it draws on it. This avoids scrolling the screen, making it easier to read while *more* is writing. This option will be ignored if the terminal does not have the ability to clear to the end of a line.

−d *More* will prompt the user with the message "Hit space to continue, Rubout to abort" at the end of each screenful. This is useful if *more* is being used as a filter in some setting, such as a class, where many users may be unsophisticated.

−f This causes *more* to count logical, rather than screen lines. That is, long lines are not folded. This option is recommended if *nroff* output is being piped through *ul*, since the latter may generate escape sequences. These escape sequences contain characters which would ordinarily occupy screen postions, but which do not print when they are sent to the terminal as part of an escape sequence. Thus *more* may think that lines are longer than they actually are, and fold lines erroneously.

−l Do not treat ^L (form feed) specially. If this option is not given, *more* will pause after any line that contains a ^L, as if the end of a screenful had been reached. Also, if a file begins with a form feed, the screen will be cleared before the file is printed.

−s Squeeze multiple blank lines from the output, producing only one blank line. Especially helpful when viewing *nroff* output, this option maximizes the useful information present on the screen.

−u Normally, *more* will handle underlining such as produced by *nroff* in a manner appropriate to the particular terminal: if the terminal can perform underlining or has a stand-out mode, *more* will output appropriate escape sequences to enable underlining or stand-out mode for underlined information in the source file. The −*u* option suppresses this processing.

+*linenumber*
Start up at *linenumber*.

+/*pattern*
Start up two lines before the line containing the regular expression *pattern*.

If the program is invoked as *page*, then the screen is cleared before each screenful is printed (but only if a full screenful is being printed), and $k - 1$ rather than $k - 2$ lines are printed in each screenful, where k is the number of lines the terminal can display.

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

More looks in the environment variable *MORE* to pre-set any flags desired. For example, if you prefer to view files using the −*c* mode of operation, the *csh* command *setenv MORE -c* or the *sh* command sequence *MORE='-c' ; export MORE* would cause all invocations of *more* , including invocations by programs such as *man* and *msgs* , to use this mode. Normally, the user will place the command sequence which sets up the *MORE* environment variable in the *.cshrc* or *.profile* file.

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

Other sequences which may be typed when *more* pauses, and their effects, are as follows (*i* is an optional integer argument, defaulting to 1) :

i<space>
: display *i* more lines, (or another screenful if no argument is given)

^D
: display 11 more lines (a "scroll"). If *i* is given, then the scroll size is set to *i*.

d
: same as ^D (control-D)

*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

q or Q
: Exit from *more*.

=
: Display the current line number.

v
: Start up the editor *vi* at the current line.

h
: Help command; give a description of all the *more* commands.

i/expr
: search for the *i*-th occurrence of the regular expression *expr*. If there are less than *i* occurrences of *expr*, and the input is a file (rather than a pipe), then the position in the file remains unchanged. Otherwise, a screenful is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the regular expression. Erasing back past the first column cancels the search command.

*i*n
: search for the *i*-th occurrence of the last regular expression entered.

'
: (single quote) Go to the point from which the last search started. If no search has been performed in the current file, this command goes back to the beginning of the file.

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

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

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

:f
: display the current file name and line number.

:q or :Q
: exit from *more* (same as q or Q).

. (dot) repeat the previous command.

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

At any time when output is being sent to the terminal, the user can hit the quit key (normally control−\). *More* will stop sending output, and will display the usual --More-- prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

The terminal is set to *noecho* mode by this program so that the output can be continuous. What you type will thus not show on your terminal, except for the / 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).

A sample usage of *more* in previewing *nroff* output would be

    nroff −ms +2 doc.n | more -s

AUTHOR

Eric Shienbrood, minor revisions by John Foderaro and Geoffrey Peck

FILES

/etc/termcap	Terminal data base
/usr/lib/more.help	Help file

SEE ALSO

csh(1), man(1), msgs(1), script(1), sh(1), environ(5)

NAME

msgs − system messages and junk mail program

SYNOPSIS

msgs [**−fhlpq**] [number] [−number]

DESCRIPTION

Msgs is used to read system messages. These messages are sent by mailing to the login 'msgs' and should be short pieces of information which are suitable to be read once by most users of the system.

Msgs is normally invoked each time you login, by placing it in the file *.login (.profile* if you use */bin/sh*). It will then prompt you with the source and subject of each new message. If there is no subject line, the first few non-blank lines of the message will be displayed. If there is more to the message, you will be told how long it is and asked whether you wish to see the rest of the message. The possible responses are:

y type the rest of the message

RETURN
synonym for y.

n skip this message and go on to the next message.

− redisplay the last message.

q drops you out of *msgs;* the next time you run the program it will pick up where you left off.

s append the current message to the file "Messages" in the current directory; 's−' will save the previously displayed message. A 's' or 's−' may be followed by a space and a filename to receive the message replacing the default "Messages".

m or 'm−' causes a copy of the specified message to be placed in a temporary mailbox and *mail*(1) to be invoked on that mailbox. Both 'm' and 's' accept a numeric argument in place of the '−'.

Msgs keeps track of the next message you will see by a number in the file *.msgsrc* in your home directory. In the directory */usr/msgs* it keeps a set of files whose names are the (sequential) numbers of the messages they represent. The file */usr/msgs/bounds* shows the low and high number of the messages in the directory so that *msgs* can quickly determine if there are no messages for you. If the contents of *bounds* is incorrect it can be fixed by removing it; *msgs* will make a new *bounds* file the next time it is run.

Options to msgs include:

−f which causes it not to say "No new messages.". This is useful in your *.login* file since this is often the case here.

−q Queries whether there are messages, printing "There are new messages." if there are. The command "msgs −q" is often used in login scripts.

−h causes *msgs* to print the first part of messages only.

−l option causes only locally originated messages to be reported.

num A message number can be given on the command line, causing *msgs* to start at the specified message rather than at the next message indicated by your *.msgsrc* file. Thus

msgs −h 1

prints the first part of all messages.

−number
will cause *msgs* to start *number* messages back from the one indicated by your *.msgsrc*

file, useful for reviews of recent messages.

−p causes long messages to be piped through *more*(1).

Within *msgs* you can also go to any specific message by typing its number when *msgs* requests input as to what to do.

FILES

/usr/msgs/*	database
~/.msgsrc	number of next message to be presented

AUTHORS

William Joy
David Wasley

SEE ALSO

mail(1), more(1)

BUGS

NAME

mt − magnetic tape manipulating program

SYNOPSIS

mt [**−t** *tapename*] *command* [*count*]

DESCRIPTION

Mt is used to give commands to the tape drive. If a tape name is not specified, /dev/rmt12 is used. If a count is not specified, 1 is assumed.

Here are the commands:

eof	write *count* end-of-file marks
fsf	space forward *count* files
fsr	space forward *count* records
bsf	space backward *count* files
bsr	space backward *count* records
rew	rewind tape
offl	rewind tape and go offline:

FILES

/dev/rmt* Raw magnetic tape interface

SEE ALSO

mt(4), dd(1)

BUGS

NAME

mv − move or rename files

SYNOPSIS

mv [**−i**] [**−f**] [**−**] file1 file2

mv [**−i**] [**−f**] [**−**] file ... directory

DESCRIPTION

Mv moves (changes the name of) *file1* to *file2.*

If *file2* already exists, it is removed before *file1* is moved. If *file2* has a mode which forbids writing, *mv* prints the mode (see *chmod*(2)) and reads the standard input to obtain a line; if the line begins with y, the move takes place; if not, *mv* exits.

In the second form, one or more *files* are moved to the *directory* with their original file-names.

Mv refuses to move a file onto itself.

Options:

−i stands for interactive mode. Whenever a move is to supercede an existing file, the user is prompted by the name of the file followed by a question mark. If he answers with a line starting with 'y', the move continues. Any other reply prevents the move from occurring.

−f stands for force. This option overrides any mode restrictions or the −i switch.

− means interpret all the following arguments to *mv* as file names. This allows file names starting with minus.

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.

Directories may only be moved within the same parent directory.

NAME

net − execute a command on a remote machine

SYNOPSIS

net [**−m** machine] [**−l** login] [**−p** password] [**−r** respfile] [**−**] [**−f**] [**−n**] [**−q**] command

DESCRIPTION

The *net* command sends the specified *command* (which should be enclosed in quotes) over the network to the specified (or default) remote machine. The network will notify the user when the command has been executed and will return to him any output or error indication by 'writing' (see *write*(1)) to the terminal if he is still logged in, or 'mailing' (see *mail*(1)) otherwise.

There are a number of options, which must precede the command. Options may be specified on the command line, preceding the command, or in a file ".netrc" in the user's login directory. The ".netrc" file is not described here. The **−m** option specifies the desired remote machine. If a remote machine is not specified, the default one is used. The machine name may be a one letter abbreviation or a full name; upper− and lower−case distinctions are ignored. If the standard output and standard error files are to be saved, the **−r** option returns to the originating user a file *(respfile)* containing the standard output and error files when the command was executed on the remote machine. If this option is used, no message is written back. The presence of a non-zero length *respfile* indicates completion. The **−q** option suppresses all acknowledgements unless an error occurs, there is output from the command, or the exit code of *command* is non-zero.

If the **−l** and **−p** options are not specified, and the login name and password are not in the ".netrc" file, a remote login name and password is prompted for on the terminal; the **−f** flag forces login name and password prompting. A single **−** indicates that the standard input from the local machine is to be taken and transmitted to the remote machine, where it will be the standard input for *command.* The **−n** flag forces all acknowledgment and output messages to be mailed rather than written on the terminal. Options do not need to be separated by spaces, i.e. either "−m C" or "−mC" is accepted. There are also other options intended to be used by higher level application programs and shell scripts only; they will not be described here.

The net command prepares a file to be sent to the remote machine and queues it in the 'network queue.' *Netq* (1) gives information about the queues.

AUTHOR

Eric Schmidt

FILES

/usr/spool/berknet/logfile	logfile with information about net activity
/usr/spool/berknet/plogfile?	log file including packet transmission statistics
/usr/spool/berknet/netstat?	statistics file
/usr/net/network.map	local network names and topology

BUGS

−q should be the default.

SEE ALSO

netrm(1), netq(1), netlog(1), netcp(1), netlpr(1), netmail(1), netlogin(1), mail(1)
"An Introduction to the Berkeley Network", by Eric Schmidt

NAME

netcp − remote copy of files through the net

SYNOPSIS

netcp [**−l** login] [**−p** password] [**−f**] [**−n**] [**−q**] fromfile tofile

DESCRIPTION

Netcp copies files between machines and is similar to *cp*(1). At least one of *fromfile* and *tofile* must be remote. The **−l, −p, −f, −q,** and **−n** behave exactly as in *net*(1).

Fromfile and *tofile* follow these conventions:

1. A simple filename is assumed to be local and from the current directory.
2. A filename preceded by a machine designator (see below) is a reference to a file on the specified remote machine. If a full pathname is not given, it is assumed to be from the login directory.

Examples:

grades.p	file in the current directory on local machine
C:junk	file in your login directory on C
/usr/lib/pq	file on local machine
C:comp/c2.c	file in a subdirectory on C machine

When files are being "fetched", that is, the *fromfile* is remote and the *tofile* is local, the *tofile* is created zero-length mode 600. For security reasons, when the "fetched" file's contents arrive at the local machine, the file must still be zero-length and mode 0600. No confirmation is sent to the user that the file has been "fetched"; a non-zero file length indicates completion.

Netcp executes the *net*(1) command.

SEE ALSO

net(1), netrm(1), netq(1), netlog(1), netlpr(1), netmail(1), netlogin(1), cp(1), mail(1)

AUTHOR

Eric Schmidt

BUGS

The second filename may not be defaulted to a directory name as in *cp*(1), it must be given explicitly.
The file mode may or may not be set correctly.

NAME

netlog − print the last few lines of the network log file

SYNOPSIS

netlog [−lines]

DESCRIPTION

Netlog prints the last few lines of the network log file indicating recent network activity.

FILES

/usr/spool/berknet/logfile	the log

SEE ALSO

net(1), netrm(1), netq(1), netcp(1), netlpr(1), netmail(1), netlogin(1), mail(1)

BUGS

NAME

netlogin — provide login name and password for a remote machine

SYNOPSIS

netlogin **−m** machine [**−l** login]

DESCRIPTION

The *netlogin* command sets the login name and password for the specified *machine* in a rather unusual way. The user should type (to the C shell)

setenv MACH*machine* `netlogin −m *machine*`

or (to the default Version 7 "Bourne" shell)

MACH*machine*=`netlogin −m *machine*`; export MACH*machine*

to his login shell. (Note the back-quotes). For example,

setenv MACHA `netlogin −m A`

will prompt the user for his login name and password on the A machine and

setenv MACHA `netlogin −m A −l myname`

will prompt the user for the password to account 'A:myname'.

The *net*(1) command will read the environment looking for environment variables beginning with "MACH" and followed by a valid machine name on the local network. If found it will use that information rather than prompt the user every time he executes a network command. This environment information is ignored if login names and passwords are specified on the command line of network commands using the **−l** and **−p** options or in the *.netrc* file.

This procedure for specifying passwords is somewhat safer than putting the remote passwords in the *.netrc* file. The passwords in the environment are encrypted and the environment information is useless after the user logs out. Use the *printenv*(1) command to see the encrypted password.

AUTHOR

Eric Schmidt

SEE ALSO

net(1), netrm(1), netq(1), netlog(1), netcp(1), netlpr(1), netmail(1), printenv(1), csh(1)

BUGS

NAME

netlpr − use a remote lineprinter through the net

SYNOPSIS

netlpr [**−m** machine] [**−l** login] [**−p** password] [**−f**] [**−q**] [**−n**] [**−c** command] [name1 ... namen]

DESCRIPTION

Netlpr sends the named files, (or the standard input if none are named), to a remote lineprinter; the **−m** option forces the files to be printed on the specified machine. (If not specified, the default machine is used.) The **−l**, **−p**, **−f**, **−q**, and **−n** options behave exactly as in *net*(1). If the **−c** option is specified, the *command* is used in place of 'lpr'. This allows the use of different lineprinters on the remote machine. See the file *'/usr/net/network.map'* for a list of available commands. Any other options are passed through to *lpr*(1) on the remote machine. Copies of the files are not made on the remote machine.

Netlpr executes the *net*(1) command.

FILES

/usr/net/network.map lists the allowed local printer names

SEE ALSO

net(1), netrm(1), netq(1), netlog(1), netcp(1), netmail(1), netlogin(1), mail(1), lpr(1)

AUTHOR

Eric Schmidt

NAME

netmail − read mail on a remote machine over the network

SYNOPSIS

netmail [**−l** username] [**−p** password] [**−c**] [**−q**] [**−n**] [**−f**] [machine:username]

DESCRIPTION

Mail is checked and/or read on the specified *machine.* If the machine specification is omitted, the default machine is used. The command has two distinct modes depending on whether the **−c** option is specified.

If **−c** is specified, the presence of mail is checked on the remote machine. No password is required so it can be put in C shell '.netrc' file. A message is written or mailed back (see *net*(1)) if there is or is not any unread mail.

If the **−c** option is not specified, mail is read and mailed back to the user. A password is required. Mail is also appended to the remote file *'mbox'* as a precaution.

The **−q** option suppresses the message sent back if there is no mail. The options **−l**, **−p**, **−f**, and **−n** behave exactly as in *net*(1). (The login name can be specified either with the **−l** option or by 'machine:username'.)

Netmail executes the *net*(1) command.

Examples:

netmail −c X:uname	checks if there is mail for 'uname' on the X machine, no password required.
netmail X:uname	reads mail for 'uname' on the X machine, mails it back, password is required.

AUTHOR

Eric Schmidt

SEE ALSO

net(1), netrm(1), netq(1), netlog(1), netcp(1), netlpr(1), netlogin(1), mail(1)

BUGS

NAME

netq − print contents of network queue

SYNOPSIS

netq [**−a**] [machine]

DESCRIPTION

Netq lists the contents of the network queue, one request per line, for each directly-connected machine. For each request, it shows the login name and machine of the originator, the destination machine and login name, and the length (in bytes) of the request (this will be larger than any files transferred (e.g. by *netcp)*, because of header information). Also described are the queue filename which may be used as an argument to *netrm*(1), the time entered the queue, and the command being sent.

Netq summarizes requests by other users. If the **−a** option is specified, requests from all users are listed.

If a *machine* is specified, only the queue for that directly-connected machine is listed.

The requests are listed in the order they will be sent; the queue for each machine is totally independent from the other machine's queues.

AUTHOR

Eric Schmidt

FILES

/usr/spool/berknet/send?	the directories where the queues are
/usr/spool/berknet/logfile	the log

SEE ALSO

net(1), netrm(1), netlog(1), netcp(1), netlpr(1), netmail(1), netlogin(1), mail(1)

BUGS

Netq should also list files in net queues on intermediate machines.
The commands are sent shortest-job first. There is no way to delay a shorter, earlier request.

NAME

netrm − remove a command from the network queue

SYNOPSIS

netrm [−] [name1 ... namen]

DESCRIPTION

Netrm removes files from the network queue which have been queued for transmission to remote machines (but not yet sent). The *names* specified are the filenames reported by the *netq*(1) command. The − option indicates that all files owned by the person logged in are to be removed.

Only the owner of the file or super-user can *netrm* the file.

AUTHOR

Eric Schmidt

FILES

/usr/spool/berknet/send? the directories where the queues are

BUGS

Files on network queues on intermediate machines cannot be removed.
There should be a **−m** flag to use with − to remove all your requests to one particular machine.

SEE ALSO

net(1), netq(1), netcp(1), netlpr(1), netmail(1), netlogin(1), mail(1)

NAME
nettroff − troff to the phototypesetter over the network

SYNOPSIS
nettroff troff arguments

DESCRIPTION
Nettroff runs *troff*(1) and sends the output over the network to the machine at the Computer Center with the phototypesetter (currently the "A" machine.) It will prompt you for an account name and password on the A machine unless you have provided these in your *.netrc* file. The *troff* −s option is unnecessary and not permitted.

Nettroff jobs are limited to 15 feet of typesetter output. It is also a good idea to limit the size of the individual files sent, as the network uses a shortest-job-first scheduling algorithm. Jobs of 25000 characters or less are preferable; in no case will the network accept jobs longer than 100000 characters.

SEE ALSO
vtroff(1), net(1), troff(1)

BUGS
There is no way to specify special font mounts on the A machine.
The **−l** and **−p** options of the *net*(1) command may not be specified.
This command is not supported by the Computer Center.
If on a Computer Center machine (B, C, D, or E) use the *troff* command (see *troff*(1)) for more information.

NAME
newaliases — rebuild the data base for the mail aliases file

SYNOPSIS
newaliases

DESCRIPTION
Newalises rebuilds the random access data base for the mail aliases file /usr/lib/aliases. It must be run each time /usr/lib/aliases is changed in order for the change to take effect.

SEE ALSO
aliases(5), delivermail(8)

BUGS

NAME

newcsh − description of new csh features (over oldcsh)

SYNOPSIS

csh *csh-options*

SUMMARY

This is a summary of features new in *csh*(1) in this version of the system; an older version of *csh* is available as *oldcsh*. This newer *csh* has some new process control primitives and a few other new features. Users of *csh* must (and automatically) use the new terminal driver (summarized in *newtty(4)* and completely described with the old in *tty*(4)) which allows generation of some new interrupt signals from the keyboard which tell jobs to stop, and arbitrates access to the terminal; on CRT's the command "stty crt" is normally placed in the *.login* file to be executed at login, to set other useful modes of this terminal driver.

Jobs.

The most important new feature in this shell is the control of *jobs*. A job is associated with each pipeline, where a pipeline is either a simple command like "date", or a pipeline like "who | wc". The shell keeps a table of current jobs, and assigns them small integer numbers. When you start a job in the background, the shell prints a line which looks like:

[1] 1234

this indicating that the job which was started asynchronously with "&" is job number 1 and has one (top-level) process, whose process id is 1234. The set of current jobs is listed by the *jobs* command.

If you are running a job and wish to do something else you may hit the key ^Z (control-Z) which sends a *stop* signal to the current job. The shell will then normally indicate that the job has been "Stopped", and print another prompt. You can then put the job in the background with the command "bg", or run some other commands and then return the job to the foreground with "fg". A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is another special key ^Y which does not generate a stop signal until a program attempts to *read*(2) it. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after it has read them.

A job being run in the background will stop if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by doing "stty tostop". If you set this tty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character "%" introduces a job name. If you wish to refer to job number 1, you can name it as "%1". Just naming a job brings it to the foreground; thus "%1" is a synonym for "fg %1", bringing job 1 back into the foreground. Similarly saying "%1 &" resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them, if these prefixes are unambiguous, thus "%ex" would normally restart a suspended *ex*(1) job, if there were only one suspended job whose name began with the string "ex". It is also possible to say "%?string" which specifies a job whose text contains *string*, if there is only one such job.

The shell also maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a "+" and the previous job with a "−". The abbreviation "%+" refers to the current job and "%−" refers to the previous job. For close analogy with the *history* mechanism, "%%" is also a synonym for the current job.

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. If, however, you set the shell variable *notify*, the shell will notify you immediately of changes of status in background jobs. There is also a shell command *notify* which marks a single process so that its status changes will be immediately reported. By default *notify* marks the current process; simply say "notify" after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that "You have stopped jobs." You may use the "jobs" command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be unmercifully terminated.

New builtin commands.

bg
bg %job ...
: Puts the current or specified jobs into the background, continuing them if they were stopped.

fg
fg %job ...
: Brings the current or specified jobs into the foreground, continuing them if they were stopped.

jobs
jobs −l
: Lists the active jobs; given the **−l** options lists process id's in addition to the normal information.

kill %job
kill −sig %job ...
kill pid
kill −sig pid ...
kill −l
: Sends either the TERM (terminate) signal or the specified signal to the specified jobs or 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 job. 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.

notify
notify %job ...
: Causes the shell to notify the user asynchronously when the status of the current or specified jobs changes; normally notification is presented before a prompt. All jobs are marked "notify" if the shell variable "notify" is set.

stop %job ...
: Stops the specified job which is executing in the background.

%job
: Brings the specified job into the foreground.

%job &
: Continues the specified job in the background.

Process limitations.

The shell provides access to an experimental facility for limiting the consumption by a single process of system resources. The following commands control this facility:

limit *resource maximum-use*

limit *resource*
limit

Limits the consumption by the current process and each process it creates to not individually exceed *maximum-use* on the specified *resource.* If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given.

Resources controllable currently include *cputime* (the maximum number of cpu-seconds to be used by each process), *filesize* (the largest single file which can be created), *datasize* (the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text), *stacksize* (the maximum size of the automatically-extended stack region), and *coredumpsize* (the size of the largest core dump that will be created).

The *maximum-use* may be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is "k" or "kilobytes" (1024 bytes); a scale factor of "m" or "megabytes" may also be used. For cputime the default scaling is "seconds", while "m" for minutes or "h" for hours, or a time of the form "mm:ss" giving minutes and seconds may be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

unlimit *resource*
unlimit

Removes the limitation on *resource.* If no *resource* is specified, then all *resource* limitations are removed.

Directory stack.

This shell now keeps track of the current directory (which is kept in the variable *cwd)* and also maintains a stack of directories, which is printed by the command *dirs.* You can change to a new directory and push down the old directory stack by using the command *pushd* which is otherwise like the *chdir* command, changing to its argument. You can pop the directory stack by saying *popd.* Saying *pushd* with no arguments exchanges the top two elements of the directory stack. The elements of the directory stack are numbered from 1 starting at the top. Saying *pushd* with a argument "+*n*" rotates the directory stack to make that entry in the stack be at the top and changes to it. Giving *popd* a "+*n*" argument eliminates that argument from the directory stack.

Miscellaneous.

This shell imports the environment variable USER into the variable *user,* TERM into *term,* and HOME into *home,* and exports 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. (It could be set once in the *.login* except that commands over the Berknet would not see the definition.)

There are new commands *eval,* which is like the eval of the Bourne shell *sh*(1), and useful with *tset*(1), and *suspend* which stops a shell (as though a ^Z had stopped it; since shells normally ignore ^Z signals, this command is necessary.)

There is a new variable *cdpath:* if set, then each directory in *cdpath* will be searched for a directory named in a *chdir* command if there is no such subdirectory of the current directory.

An *unsetenv* command removing environment variables has been added.

There is a new ":" modifier ":e", which yields the extension portion of a filename. Thus if "$a" is "file.c", "$a:e" is "c".

There are two new operators in shell expressions "!~" and "=~" which are like the string operations "!=" 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.

The form "$<" is new, and is replaced by a line from the standard input, with no further interpretation thereafter. It may therefore be used to read from the keyboard in a shell script.

SEE ALSO

csh(1), killpg(2), sigsys(2), signal(2), jobs(3), sigset(3), tty(4)

BUGS

Command sequences of the form "a ; b ; c" are not handled gracefully when stopping is attempted. If you suspend "b", the shell will then immediately execute "c". This is especially noticeable if this expansion results from an *alias*. It suffices to place the sequence of commands in ()'s to force it to a subshell, i.e. "(a ; b ; c)", but see the next bug.

Shell builtin functions are not stoppable/restartable.

Control over output 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.

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)

BUGS

NAME

nice, nohup − run a command at low priority (*sh* only)

SYNOPSIS

nice [−*number*] command [arguments]

nohup command [arguments]

DESCRIPTION

Nice executes *command* with low scheduling priority. If the *number* argument is present, the priority is incremented (higher numbers mean lower priorities) by that amount up to a limit of 20. The default *number* is 10.

The super-user may run commands with priority higher than normal by using a negative priority, e.g. '− −10'.

Nohup executes *command* immune to hangup and terminate signals from the controlling terminal. The priority is incremented by 5. *Nohup* should be invoked from the shell with '&' in order to prevent it from responding to interrupts by or stealing the input from the next person who logs in on the same terminal. The syntax of nice is also different.

FILES

nohup.out standard output and standard error file under *nohup*

SEE ALSO

csh(1), nice(2), renice(8)

DIAGNOSTICS

Nice returns the exit status of the subject command.

BUGS

Nice and *nohup* are particular to *sh*(1). If you use *csh*(1), then commands executed with "&" are automatically immune to hangup signals while in the background. There is a builtin command *nohup* which provides immunity from terminate, but it does not redirect output to *nohup.out*.

Nice is built into *csh*(1) with a slightly different syntax than described here. The form "nice +10" nices to positive nice, and "nice −10" can be used by the super-user to give a process more of the processor.

NAME

nm — print name list

SYNOPSIS

nm [**−agnopru**] [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, or − for sdb symbol table entries (see **−a** below). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

Options are:

−a Include all symbols in candidates for printing; normally symbols destined for *sdb*(1) are excluded.

−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.

SEE ALSO

ar(1), ar(5), a.out(5), stab(5)

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.

Num is actually just the **−n** option of the *cat*(1) command.

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 continues until end-of-file.

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.

New passwords must be at least four characters long if they use a sufficiently rich alphabet and at least six characters long if monocase. These rules are relaxed if you are insistent enough.

Only the owner of the name or the super-user may change a password; the owner must prove he knows the old password.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(5), crypt(3)
Robert Morris and Ken Thompson, *UNIX password security*

NAME

pc — Pascal compiler

SYNOPSIS

pc [option] [**−i** name ...] name ...

DESCRIPTION

Pc is a Pascal compiler. If given an argument file ending with **.p**, it will compile the file and load it into an executable file called, by default, *a.out*.

A program may be separated into more than one **.p** file. *Pc* will compile a number of argument **.p** files into object files (with the extension **.o** in place of **.p**). Object files may then be loaded into an executable *a.out* file. Exactly one object file must supply a **program** statement to successfully create an executable a.out file. The rest of the files must consist only of declarations which logically nest within the program. References to objects shared between separately compiled files are allowed if the objects are declared in **included** header files, whose names must end with **.h**. Header files may only be included at the outermost level, and thus declare only globally available objects. To allow **functions** and **procedures** to be declared, an **external** directive has been added, whose use is similar to the **forward** directive but restricted to appear only in **.h** files. **Function** and **procedure** bodies may not appear in **.h** files. A binding phase of the compiler checks that declarations are used consistently, to enforce the type checking rules of Pascal.

Object files created by other language processors may be loaded together with object files created by *pc*. The **functions** and **procedures** they define must have been declared in **.h** files included by all the **.p** files which call those routines. Calling conventions are as in C, with **var** parameters passed by address.

See the Berkeley Pascal User's Manual for details.

The following options have the same meaning as in *cc*(1) and *f77*(1). See *ld*(1) for load-time options.

−c Suppress loading and produce '.o' file(s) from source file(s).

−g Have the compiler produce additional symbol table information for *sdb*(1).

−w Suppress warning messages.

−p Prepare object files for profiling, see *prof*(1).

−O Invoke an object-code improver.

−S Compile the named program, and leave the assembler-language output on the corresponding file suffixed '.s'. (No '.o' is created.).

−o output
Name the final output file *output* instead of *a.out*.

The following options are peculiar to *pc*.

−C Compile code to perform runtime checks, verify **assert** statements, and initialize all variables to zero as in *pi*.

−b Block buffer the file *output*.

−i Produce a listing for the specified procedures, functions and **include** files.

−l Make a program listing during translation.

−s Accept standard Pascal only; non-standard constructs cause warning diagnostics.

−z Allow execution profiling with *pxp* by generating statement counters, and arranging for the creation of the profile data file *pmon.out* when the resulting object is executed.

Other arguments are taken to be loader option arguments, perhaps libraries of *pc* compatible routines. Certain flags can also be controlled in comments within the program as described in the *Berkeley Pascal User's Manual.*

FILES

file.p	pascal source files
/usr/lib/pc0	compiler
/lib/f1	code generator
/usr/lib/pc2	runtime integrator (inline expander)
/lib/c2	peephole optimizer
/usr/lib/pc3	separate compilation consistency checker
/usr/lib/pc2.0strings	text of the error messages
/usr/lib/how_pc	basic usage explanation
/usr/lib/libpc.a	intrinsic functions and I/O library
/usr/lib/libm.a	math library
/lib/libc.a	standard library, see *intro*(3)

SEE ALSO

Berkeley Pascal User's Manual
pi(1), pxp(1), pxref(1), sdb(1)

DIAGNOSTICS

For a basic explanation do

pc

See *pi*(1). for an explanation of the error message format. Internal errors cause messages containing the word SNARK.

AUTHORS

Charles B. Haley, William N. Joy, and Ken Thompson
Retargetted to the second pass of the portable *C* compiler by Peter Kessler
Runtime library and inline optimizer by M. Kirk McKusick
Separate compilation consistency checking by Louise Madrid

BUGS

The keyword **packed** is recognized but has no effect.

The binder is not as strict as described here, with regard to the rules about external declarations only in '.h' files and including '.h' files only at the outermost level. It will be made to perform these checks in its next incarnation, so users are warned not to be sloppy.

The **−z** flag doesn't work for separately compiled files.

Because the **−s** option is usurped by the compiler, it is not possible to pass the strip option to the loader. Thus programs which are to be stripped, must be run through *strip*(1) after the are compiled.

NAME

pi — Pascal interpreter code translator

SYNOPSIS

pi [−blnpstuwz] [−i name ...] name.p

DESCRIPTION

Pi translates the program in the file *name.p* leaving interpreter code in the file *obj* in the current directory. The interpreter code can be executed using *px*. *Pix* performs the functions of *pi* and *px* for 'load and go' Pascal.

The following flags are interpreted by *pi;* the associated options can also be controlled in comments within the program as described in the *Berkeley Pascal User's Manual.*

−b Block buffer the file *output.*

−i Enable the listing for any specified procedures and functions and while processing any specified **include** files.

−l Make a program listing during translation.

−n Begin each listed **include** file on a new page with a banner line.

−p Suppress the post-mortem control flow backtrace if an error occurs; suppress statement limit counting.

−s Accept standard Pascal only; non-standard constructs cause warning diagnostics.

−t Suppress runtime tests of subrange variables and treat **assert** statements as comments.

−u Card image mode; only the first 72 characters of input lines are used.

−w Suppress warning diagnostics.

−z Allow execution profiling with *pxp* by generating statement counters, and arranging for the creation of the profile data file *pmon.out* when the resulting object is executed.

FILES

file.p	input file
file.i	include file(s)
/usr/lib/pi_strings	text of the error messages
/usr/lib/how_pi*	basic usage explanation
obj	interpreter code output

SEE ALSO

Berkeley Pascal User's Manual
pix(1), px(1), pxp(1), pxref(1)

DIAGNOSTICS

For a basic explanation do

pi

In the diagnostic output of the translator, lines containing syntax errors are listed with a flag indicating the point of error. Diagnostic messages indicate the action which the recovery mechanism took in order to be able to continue parsing. Some diagnostics indicate only that the input is 'malformed.' This occurs if the recovery can find no simple correction to make the input syntactically valid.

Semantic error diagnostics indicate a line in the source text near the point of error. Some errors evoke more than one diagnostic to help pinpoint the error; the follow-up messages begin with an ellipsis '...'.

The first character of each error message indicates its class:

E	Fatal error; no code will be generated.
e	Non-fatal error.
w	Warning — a potential problem.
s	Non-standard Pascal construct warning.

If a severe error occurs which inhibits further processing, the translator will give a diagnostic and then 'QUIT'.

AUTHORS

Charles B. Haley, William N. Joy, and Ken Thompson
Ported to VAX-11 by Peter Kessler

BUGS

Formal parameters which are procedures and functions are not supported.

The keyword **packed** and the function *dispose* are recognized but have no effect.

For clarity, semantic errors should be flagged at an appropriate place in the source text, and multiple instances of the 'same' semantic error should be summarized at the end of a **procedure** or **function** rather than evoking many diagnostics.

When **include** files are present, diagnostics relating to the last procedure in one file may appear after the beginning of the listing of the next.

NAME
pix − Pascal interpreter and executor

SYNOPSIS
pix [**−blnpstuwz**] [**−i** name ...] name.p [argument ...]

DESCRIPTION
Pix is a 'load and go' version of Pascal which combines the functions of the interpreter code translator *pi* and the executor *px*. It uses *pi* to translate the program in the file *name.p* and, if there were no fatal errors during translation, causes the resulting interpreter code to be executed by *px* with the specified arguments. A temporary file is used for the object code; the file *obj* is neither created nor destroyed.

FILES

/usr/bin/pi	Pascal translator
/usr/bin/px	Pascal executor
/tmp/pix?????	temporary
/usr/lib/how_pix	basic explanation

SEE ALSO
Berkeley Pascal User's Manual
pi(1), px(1)

DIAGNOSTICS
For a basic explanation do

pix

AUTHORS
Susan L. Graham and William N. Joy

NAME
plot − graphics filters

SYNOPSIS
plot [−Tterminal [raster]]

DESCRIPTION
These commands read plotting instructions (see *plot*(5)) from the standard input, and in general produce plotting instructions suitable for a particular *terminal* on the standard output.

If no *terminal* type is specified, the environment parameter $TERM (see *environ*(5)) is used. Known *terminals* are:

4014 Tektronix 4014 storage scope.

450 DASI Hyterm 450 terminal (Diablo mechanism).

300 DASI 300 or GSI terminal (Diablo mechanism).

300S DASI 300S terminal (Diablo mechanism).

ver Versatec D1200A printer-plotter. This version of *plot* places a scan-converted image in '/usr/tmp/raster' and sends the result directly to the plotter device rather than to the standard output. The optional argument causes a previously scan-converted file *raster* to be sent to the plotter.

FILES
/usr/bin/tek
/usr/bin/t450
/usr/bin/t300
/usr/bin/t300s
/usr/bin/vplot
/usr/tmp/raster

SEE ALSO
plot(3), plot(5)

BUGS
There is no lockout protection for /usr/tmp/raster.

NAME
pmerge – pascal file merger

SYNOPSIS
pmerge name.p ...

DESCRIPTION
Pmerge assembles the named Pascal files into a single standard Pascal program. The resulting program is listed on the standard output. It is intended to be used to merge a collection of separately compiled modules so that they can be run through **pi** , or exported to other sites.

FILES
/usr/tmp/MG* default temporary files

SEE ALSO
pc(1), pi(1),
Auxiliary documentation *Berkeley Pascal User's Manual.*

AUTHOR
M. Kirk McKusick

BUGS
Very minimal error checking is done, so incorrect programs will produce unpredictable results. Block comments should be placed after the keyword to which they refer or they are likely to end up in bizarre places.

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.*

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 [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a *name* is specified, only its value is printed.

If a *name* is specified and it is not defined in the environment, *printenv* returns exit status 1, else it returns status 0.

SEE ALSO

sh(1), environ(5), csh(1)

BUGS

NAME

prmail – print out mail in the post office

SYNOPSIS

prmail [user ...]

DESCRIPTION

Prmail prints the mail which waits for you, or the specified user, in the post office. The mail is not disturbed.

FILES

/usr/spool/mail/* post office

SEE ALSO

biff(1), mail(1), from(1), binmail(1)

BUGS

NAME

prof − display profile data

SYNOPSIS

prof [**−a**] [**−l**] [**−n**] [**−z**] [**−s**] [**−v** [**−***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, f77* or *pc* 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.

−n the output is sorted by number of calls

−s a summary profile file is produced in *mon.sum.* This is really only useful when more than one profile file is specified.

−v all printing is suppressed and a graphic version of the profile is produced on the standard output for display by the *plot*(1) filters. When plotting, the numbers *low* and *high,* by default 0 and 100, may be given to cause a selected percentage of the profile to be plotted with accordingly higher resolution.

−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

monitor(3), profil(2), cc(1), plot(1)

BUGS

Beware of quantization errors.

Is confused by *f77* which puts the entry points at the bottom of subroutines and functions.

NAME

ps − process status

SYNOPSIS

ps [acegklstuvwx#]

DESCRIPTION

Ps prints information about processes. Normally, only your processes are candidates to be printed by *ps;* specifying **a** causes other users processes to be candidates to be printed; specifying **x** includes processes without control terminals in the candidate pool.

All output formats include, for each process, the process id PID, control terminal of the process TT, cpu time used by the process TIME (this includes both user and system time), the state STAT of the process, and an indication of the COMMAND which is running. The state is given by a sequence of four letters, e.g. "RWNA". The first letter indicates the runnability of the process: R for runnable processes, T for stopped processes, P for processes in page wait, D for those in disk (or other short term) waits, S for those sleeping for less than about 20 seconds, and I for idle (sleeping longer than about 20 seconds) processes. The second letter indicates whether a process is swapped out, showing W if it is, or a blank if it is loaded (in-core); a process which has specified a soft limit on memory requirements and which is exceeding that limit shows >; such a process is (necessarily) not swapped. The third letter indicates whether a process is running with altered CPU scheduling priority (nice); if the processes priority is reduced, a N is shown, if the process priority has been artificially raised then a '<' is shown; process running without special treatment have just a blank. The final letter indicates any special treatment of the process for virtual memory replacement; the letters correspond to options to the *vadvise*(2) call; currently the possibilities are A standing for VA_ANOM, S for VA_SEQL and blank for VA_NORM; an A typically represents a *lisp*(1) in garbage collection, S is typical of large image processing programs which are using virtual memory to sequentially address voluminous data.

Here are the options:

a asks for information about all processes with terminals (ordinarily only one's own processes are displayed).

c prints the command name, as stored internally in the system for purposes of accounting, rather than the command arguments, which are kept in the process' address space. This is more reliable, if less informative, since the process is free to destroy the latter information.

e Asks for the environment to be printed as well as the arguments to the command.

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. This normally eliminates top-level command interpreters and processes waiting for users to login on free terminals.

k causes the file */vmcore* is used in place of */dev/kmem* and */dev/mem.* This is used for post-mortem system debugging.

l asks for a long listing, with fields PPID, CP, PRI, NI, ADDR, SIZE, RSS and WCHAN as described below.

s Adds the size SSIZ of the kernel stack of each process (for use by system maintainers) to the basic output format.

t*x* restricts output to processes whose controlling tty is *x* (which should be specified as printed by *ps,* e.g. *t3* for tty3, *tco* for console, *td0* for ttyd0, *t?* for processes with no tty, etc). This option must be the last one given.

u A user oriented output is produced. This includes fields USER, %CPU, NICE, SIZE, and

RSS as described below.

v A version of the output containing virtual memory statistics is output. This includes fields RE, SL, PAGEIN, SIZE, RSS, LIM, TSIZ, TRS, %CPU and %MEM, described below.

w Use a wide output format (132 columns rather than 80); if repeated, e.g. ww, use arbitrarily wide output. This information is used to decide how much of long commands to print.

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 /vmcore. A third argument is the name of a swap file to use instead of the default /dev/drum. If a fourth argument is given, it is taken to be the file containing the system's namelist. Otherwise, /vmunix is used.

Fields which are not common to all output formats:

USER	name of the owner of the process
%CPU	cpu utilization of the process; this is a decaying average over up to a minute of previous (real) time. Since the time base over which this is computed varies (since processes may be very young) it is possible for the sum of all %CPU fields to exceed 100%.
NICE	(or NI) process scheduling increment (see *nice*(2))
SIZE	virtual size of the process (in 1024 byte units)
RSS	real memory (resident set) size of the process (in 1024 byte units)
LIM	soft limit on memory used, specified via a call to *vlimit*(2); if no limit has been specified then shown as *xx*
TSIZ	size of text (shared program) image
TRS	size of resident (real memory) set of text
%MEM	percentage of real memory used by this process.
RE	residency time of the process (seconds in core)
SL	sleep time of the process (seconds blocked)
PAGEIN	number of disk i/o's resulting from references by the process to pages not loaded in core.
UID	numerical user-id of process owner
PPID	numerical id of parent of process
CP	short-term cpu utilization factor (used in scheduling)
PRI	process priority (non-positive when in non-interruptible wait)
ADDR	swap address of the process
WCHAN	event on which process is waiting (an address in the system), with the initial part of the address trimmed off e.g. 80004000 prints as 4000.

F flags associated with process as in /usr/include/sys/proc.h:

SLOAD	000001	in core
SSYS	000002	swapper or pager process
SLOCK	000004	process being swapped out
SSWAP	000008	save area flag
STRC	000010	process is being traced
SWTED	000020	another tracing flag
SULOCK	000040	user settable lock in core
SPAGE	000080	process in page wait state
SKEEP	000100	another flag to prevent swap out

SDLYU	000200	delayed unlock of pages
SWEXIT	000400	working on exiting
SPHYSIO	000800	doing physical i/o (bio.c)
SVFORK	001000	process resulted from vfork()
SVFDONE	002000	another vfork flag
SNOVM	004000	no vm, parent in a vfork()
SPAGI	008000	init data space on demand, from inode
SANOM	010000	system detected anomalous vm behavior
SUANOM	020000	user warned of anomalous vm behavior
STIMO	040000	timing out during sleep
SDETACH	080000	detached inherited by init
SNUSIG	100000	using new signal mechanism

A process that has exited and has a parent, but has not yet been waited for by the parent is marked <defunct>; a process which is blocked trying to exit is marked <exiting>; *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

/vmunix	system namelist
/dev/kmem	kernel memory
/dev/drum	swap device
/vmcore	core file
/dev	searched to find swap device and tty names

SEE ALSO

kill(1), w(1)

BUGS

Things can change while *ps* is running; the picture it gives is only a close approximation to reality.

NAME

pti − phototypesetter interpreter

SYNOPSIS

pti [file ...]

DESCRIPTION

Pti shows the commands in a stream from the standard output of *troff*(1) using *troff's* **−t** option, interpreting them as they would act on the typesetter. Horizontal motions shows as counts in internal units and are marked with '<' and '>' indicating left and right motion. Vertical space is called *lead* and is also indicated.

SEE ALSO

troff(1)

BUGS

Too cryptic for normal users, who should use "troff −a ...".

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

pwd — working directory name

SYNOPSIS

pwd

DESCRIPTION

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

SEE ALSO

cd(1), csh(1)

BUGS

In *csh*(1) the command *dirs* is always faster (although it can give a different answer in the rare case that the current directory or a containing directory was moved after the shell descended into it).

NAME
px − Pascal interpreter

SYNOPSIS
px [obj [argument ...]]

DESCRIPTION
Px interprets the abstract machine code generated by *pi*. The first argument is the file to be interpreted, and defaults to *obj*; remaining arguments are available to the Pascal program using the built-ins *argv* and *argc*. *Px* is also invoked by *pix* when running 'load and go'.

If the program terminates abnormally an error message and a control flow backtrace are printed. The number of statements executed and total execution time are printed after normal termination. The **p** option of *pi* suppresses all of this except the message indicating the cause of abnormal termination.

FILES
obj	default object file
pmon.out	profile data file

SEE ALSO
Berkeley Pascal User's Manual
pi(1), pix(1)

DIAGNOSTICS
Most run-time error messages are self-explanatory. Some of the more unusual ones are:

Reference to an inactive file
: A file other than *input* or *output* was used before a call to *reset* or *rewrite*.

Statement count limit exceeded
: The limit of 500,000 executed statements (which prevents excessive looping or recursion) has been exceeded.

Bad data found on integer read
Bad data found on real read
: Usually, non-numeric input was found for a number. For reals, Pascal requires digits before and after the decimal point so that numbers like '.1' or '21.' evoke the second diagnostic.

panic: *Some message*
: Indicates a internal inconsistency detected in *px* probably due to a Pascal system bug.

AUTHORS
Charles B. Haley, William Joy, and Ken Thompson
VAX-11 version by Kirk McKusick

BUGS
Post-mortem traceback is not limited; infinite recursion leads to almost infinite traceback.

NAME

pxp − Pascal execution profiler

SYNOPSIS

pxp [**−acdefjnstuw_**] [−23456789] [**−z** [name ...]] name.p

DESCRIPTION

Pxp can be used to obtain execution profiles of Pascal programs or as a pretty-printer. To produce an execution profile all that is necessary is to translate the program specifying the **z** option to *pi* or *pix,* to execute the program, and to then issue the command

pxp −z name.p

A reformatted listing is output if none of the **c**, **t**, or **z** options are specified; thus

pxp old.p > new.p

places a pretty-printed version of the program in 'old.p' in the file 'new.p'.

The use of the following options of *pxp* is discussed in sections 2.6, 5.4, 5.5 and 5.10 of the *Berkeley Pascal User's Manual.*

−a Print the bodies of all procedures and functions in the profile; even those which were never executed.

−c Extract profile data from the file *core*.

−d Include declaration parts in a profile.

−e Eliminate **include** directives when reformatting a file; the **include** is replaced by the reformatted contents of the specified file.

−f Fully parenthesize expressions.

−j Left justify all procedures and functions.

−n Eject a new page as each file is included; in profiles, print a blank line at the top of the page.

−s Strip comments from the input text.

−t Print a table summarizing **procedure** and **function** call counts.

−u Card image mode; only the first 72 characters of input lines are used.

−w Suppress warning diagnostics.

−z Generate an execution profile. If no *name*s, are given the profile is of the entire program. If a list of names is given, then only any specified **procedures** or **functions** and the contents of any specified **include** files will appear in the profile.

−_ Underline keywords.

−*d* With *d* a digit, $2 \leqslant d \leqslant 9$, causes *pxp* to use *d* spaces as the basic indenting unit. The default is 4.

FILES

name.p	input file
name.i	include file(s)
pmon.out	profile data
core	profile data source with −c
/usr/lib/how_pxp	information on basic usage

SEE ALSO

Berkeley Pascal User's Manual
pi(1), px(1)

DIAGNOSTICS

For a basic explanation do

pxp

Error diagnostics include 'No profile data in file' with the **c** option if the **z** option was not enabled to *pi;* 'Not a Pascal system core file' if the core is not from a *px* execution; 'Program and count data do not correspond' if the program was changed after compilation, before profiling; or if the wrong program is specified.

AUTHOR

William Joy

BUGS

Does not place multiple statements per line.

NAME

pxref – Pascal cross-reference program

SYNOPSIS

pxref [–] name

DESCRIPTION

Pxref makes a line numbered listing and a cross-reference of identifier usage for the program in *name*. The optional '–' argument suppresses the listing. The keywords **goto** and **label** are treated as identifiers for the purpose of the cross-reference. **Include** directives are not processed, but cause the placement of an entry indexed by '#include' in the cross-reference.

SEE ALSO

Berkeley Pascal User's Manual

AUTHOR

Niklaus Wirth

BUGS

Identifiers are trimmed to 10 characters.

NAME

ranlib − convert archives to random libraries

SYNOPSIS

ranlib archive ...

DESCRIPTION

Ranlib converts each *archive* to a form which can be loaded more rapidly by the loader, by adding a table of contents named __.SYMDEF to the beginning of the archive. It uses *ar*(1) to reconstruct the archive, so that sufficient temporary file space must be available in the file system containing the current directory.

SEE ALSO

ld(1), ar(1), lorder(1)

BUGS

Because generation of a library by *ar* and randomization by *ranlib* are separate, phase errors are possible. The loader *ld* warns when the modification date of a library is more recent than the creation of its dictionary; but this means you get the warning even if you only copy the library.

NAME
ratfor − rational Fortran dialect

SYNOPSIS
ratfor [option ...] [filename ...]

DESCRIPTION
Ratfor converts a rational dialect of Fortran into ordinary irrational Fortran. *Ratfor* provides control flow constructs essentially identical to those in C:

statement grouping:
```
        { statement; statement; statement }
```

decision-making:
```
        if (condition) statement [ else statement ]
        switch (integer value) {
                case integer:   statement
                ...
                [ default: ]    statement
        }
```

loops:
```
        while (condition) statement
        for (expression; condition; expression) statement
        do limits statement
        repeat statement [ until (condition) ]
        break
        next
```

and some syntactic sugar to make programs easier to read and write:

free form input:
  multiple statements/line; automatic continuation

comments:
  # this is a comment

translation of relationals:
  >, >=, etc., become .GT., .GE., etc.

return (expression)
  returns expression to caller from function

define: define name replacement

include:
  include filename

Ratfor is best used with *f77*(1).

SEE ALSO
efl(1), f77(1)
B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

NAME

refer, lookbib − find and insert literature references in documents

SYNOPSIS

refer [option] ...

lookbib [file] ...

DESCRIPTION

Lookbib accepts keywords from the standard input and searches a bibliographic data base for references that contain those keywords anywhere in title, author, journal name, etc. Matching references are printed on the standard output. Blank lines are taken as delimiters between queries.

Refer is a preprocessor for *nroff* or *troff*(1) that finds and formats references. The input files (standard input default) are copied to the standard output, except for lines between .[and .] command lines, which are assumed to contain keywords as for *lookbib,* and are replaced by information from the bibliographic data base. The user may avoid the search, override fields from it, or add new fields. The reference data, from whatever source, are assigned to a set of *troff* strings. Macro packages such as *ms*(7) print the finished reference text from these strings. A flag is placed in the text at the point of reference; by default the references are indicated by numbers.

The following options are available:

−a*r* Reverse the first *r* author names (Jones, J. A. instead of J. A. Jones). If *r* is omitted all author names are reversed.

−b Bare mode: do not put any flags in text (neither numbers nor labels).

−c*string*
Capitalize (with CAPS SMALL CAPS) the fields whose key-letters are in *string.*

−e Instead of leaving the references where encountered, accumulate them until a sequence of the form

```
.[
$LIST$
.]
```

is encountered, and then write out all references collected so far. Collapse references to the same source.

−k*x* Instead of numbering references, use labels as specified in a reference data line beginning *%x;* by default *x* is L.

−l*m,n*
Instead of numbering references, use labels made from the senior author's last name and the year of publication. Only the first *m* letters of the last name and the last *n* digits of the date are used. If either *m* or *,n* is omitted the entire name or date respectively is used.

−p Take the next argument as a file of references to be searched. The default file is searched last.

−n Do not search the default file.

−s*keys*
Sort references by fields whose key-letters are in the *keys* string; permute reference numbers in text accordingly. Implies **−e**. The key-letters in *keys* may be followed by a number to indicate how many such fields are used, with + taken as a very large number. The default is AD which sorts on the senior author and then date; to sort, for example, on all authors and then title use **-sA+T**.

When *refer* is used with *eqn*, *neqn* or *tbl*, *refer* should be first, to minimize the volume of data passed through pipes.

FILES

/usr/dict/papers	directory of default publication lists and indexes
/usr/lib/refer	directory of programs

SEE ALSO

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

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.

NAME

rm, rmdir – remove (unlink) files

SYNOPSIS

rm [**−f**] [**−r**] [**−i**] [**−**] file ...

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission 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.

If a designated file is a directory, an error comment is printed unless the optional argument **−r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory, and the directory itself.

If the **−i** (interactive) option is in effect, *rm* asks whether to delete each file, and, under **−r**, whether to examine each directory.

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.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

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

rewind — rewind tape drive

SYNOPSIS

rewind [tape]

DESCRIPTION

Rewind rewinds the tape drive, if a tape is mounted on it. This is done by opening and closing the tape drive, using a file name that will rewind when the file is closed. If an argument is given that is taken as the name of the tape drive rather than the default /dev/mt0.

AUTHOR

Mark Horton

FILES

/dev/mt0

NAME

script — make typescript of terminal session

SYNOPSIS

script [**−a**] [**−q**] [**−S** shell] [file]

DESCRIPTION

Script makes a typescript of everything printed on your terminal. The typescript is saved in a file, and can be sent to the line printer later with *lpr*. If a file name is given, the typescript is saved there. If not, the typescript is saved in the file *typescript*.

To exit script, type control D. This sends an end of file to all processes you have started up, and causes script to exit. For this reason, control D behaves as though you had typed an infinite number of control D's.

This program is useful when using a crt and a hard-copy record of the dialog is desired, as for a student handing in a program that was developed on a crt when hard-copy terminals are in short supply.

−S lets you specify the shell to use. The default depends on the system: If the variable SHELL is set in the environment, it is used if possible.

The **−q** flag asks for "quiet mode", where the "script started" and "script done" messages are turned off. The **−a** flag causes script to append to the typescript file instead of creating a new file.

AUTHOR

Mark Horton

BUGS

Since UNIX has no way to write an end-of-file down a pipe without closing the pipe, there is no way to simulate a single control D without ending script.

The new shell has its standard input coming from a pipe rather than a tty, so stty will not work, and neither will ttyname. In particular, this means that screen editors such as *vi*(1) and the job control facilities of *csh*(1) are inoperative.

When the user interrupts a printing process, *script* attempts to flush the output backed up in the pipe for better response. Usually the next prompt also gets flushed.

NAME

sdb − symbolic debugger

SYNOPSIS

sdb [objfil [corfil [directory]]]

DESCRIPTION

Sdb is a symbolic debugger which can be used with C, PASCAL, and F77 programs. It may be used to examine their files and to provide a controlled environment for their execution.

Objfil is an executable program file which has been compiled with the -g (debug) option. 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. The core file need not be present.

It is useful to know that at any time there is a *current line* and *current file*. If *corfil* exists then they are initially set to the line and file containing the source statement at which the process terminated or stopped. Otherwise, they are set to the first line in main. The current line and file may be changed with the source file examination commands.

Names of variables are written just as they are in C, PASCAL, or F77. Variables local to a procedure may be accessed using the form 'procedure:variable'. If no procedure name is given, the procedure containing the current line is used by default. It is also possible to refer to structure members as 'variable.member', pointers to structure members as 'variable−>member' and array elements as 'variable[number]'. Combinations of these forms may also be used.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as 'filename:number' or 'procedure:number'. In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

The commands for examining data in the program are:

t Print a stack trace of the terminated or stopped program.

T Print the top line of the stack trace.

variable/*lm*

Print the value of variable according to length *l* and format *m*. If *l* and *m* are omitted, sdb chooses a length and format suitable for the variable's type as declared in the program. The length specifiers are:

- **b** one byte
- **h** two bytes (half word)
- **l** four bytes (long word)
- number
  string length for formats **s** and **a**

Legal values for *m* are:

- **c** character
- **d** decimal
- **u** decimal, unsigned
- **o** octal
- **x** hexadecimal
- **f** 32 bit single precision floating point
- **g** 64 bit double precision floating point
- **s** Assume variable is a string pointer and print characters until a null is reached.

a Print characters starting at the variable's address until a null is reached.
p pointer to procedure

The length specifiers are only effective with the formats **d**, **u**, **o** and **x**. If one of these formats is specified and *l* is omitted, the length defaults to the word length of the host machine; 4 for the DEC VAX/11-780. The last variable may be redisplayed with the command './'.

The sh(1) metacharacters * and ? may be used within procedure and variable names, providing a limited form of pattern matching. If no procedure name is given, both variables local to the current procedure and global (common for F77) variables are matched, while if a procedure name is specified then only variables local to that procedure and matched. To match only global variables (or blank common for F77), the form ':pattern' is used. The name of a common block may be specified instead of a procedure name for F77 programs.

variable=*lm*
linenumber=*lm*
number=*lm*
Print the address of the variable or line number or the value of the number in the specified format. If no format is given, then 'lx' is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

variable!value
Set the variable to the given value. The value may be a number, character constant or a variable. If the variable is of type float or double, the value may also be a floating constant.

The commands for examining source files are

e procedure
e filename.c
Set the current file to the file containing the named procedure or the named filename. Set the current line to the first line in the named procedure or file. All source files are assumed to be in *directory.* The default for *directory* is the working directory. If no procedure or file name is given, the current procedure and file names are reported.

/regular expression/
Search forward from the current line for a line containing a string matching the regular expression as in ed(1). The trailing '/' may be elided.

?regular expression?
Search backward from the current line for a line containing a string matching the regular expression as in ed(1). The trailing '?' may be elided.

p Print the current line.

z Print the current line followed by the next 9 lines. Set the current line to the last line printed.

control-D
Scroll. Print the next 10 lines. Set the current line to the last line printed.

w Window. Print the 10 lines around the current line.

number
Set the current line to the given line number. Print the new current line.

count +
Advance the current line by *count* lines. Print the new current line.

count —
Retreat the current line by *count* lines. Print the new current line.

The commands for controlling the execution of the source program are:

count r *args*
count R
Run the program with the given arguments. The r command with no arguments reuses the previous arguments to the program while the R command runs the program with no arguments. An argument beginning with '<' or '>' causes redirection for the standard input or output respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber c *count*
linenumber C *count*
Continue after a breakpoint or interrupt. If *count* is given, it specifies the number of breakpoints to be ignored. C continues with the signal which caused the program to stop and c ignores it.

If a linenumber is specified then a temporary breakpoint is placed at the line and execution is continued. The breakpoint is deleted when the command finishes.

count s
Single step. Run the program through *count* lines. If no count is given then the program is run for one line.

count S
Single step, but step through subroutine calls.

k Kill the debugged program.

procedure(arg1,arg2,...)
procedure(arg1,arg2,...)/*m*
Execute the named procedure with the given arguments. Arguments can be integer, character or string constants or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to 'd'.

linenumber b *commands*
Set a breakpoint at the given line. If a procedure name without a line number is given (e.g. 'proc:'), a breakpoint is placed at the first line in the procedure even if it was not compiled with the debug flag. If no *linenumber* is given, a breakpoint is placed at the current line.

If no *commands* are given then execution stops just before the breakpoint and control is returned to sdb. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons.

linenumber d
Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively: Each breakpoint location is printed and a line is read from the standard input. If the line begins with a 'y' or 'd' then the breakpoint is deleted.

B Print a list of the currently active breakpoints.

D Delete all breakpoints.

l Print the last executed line.

linenumber a

Announce. If *linenumber* is of the form 'proc:number', the command effectively does a 'linenumber b l'. If *linenumber* is of the form 'proc:', the command effectively does a 'proc: b T'.

Miscellaneous commands.

! *command*
The command is interpreted by sh(1).

newline
If the previous command printed a source line then advance the current line by 1 line and print the new current line. If the previous command displayed a core location then display the next core location.

" *string*
Print the given string.

q Exit the debugger.

The following commands also exist and are intended only for debugging the debugger.

V Print the version number.

X Print a list of procedures and files being debugged.

Y Toggle debug output.

FILES
a.out
core

SEE ALSO
adb(1)

DIAGNOSTICS
Error reports are either identical to those of adb(1) or are self-explanatory.

BUGS
If a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a core image.

Arrays must be of one dimension and of zero origin to be correctly addressed by sdb.

The default type for printing F77 parameters is incorrect. Their address is printed instead of their value.

Tracebacks containing F77 subprograms with multiple entry points may print too many arguments in the wrong order, but their values are correct.

Sdb understands Pascal, but not its types.

NAME

sed − stream editor

SYNOPSIS

sed [**−n**] [**−e** script] [**−f** sfile] [file] ...

DESCRIPTION

Sed copies the named *files* (standard input default) to the standard output, edited according to a script of commands. The **−f** option causes the script to be taken from file *sfile*; these options accumulate. If there is just one **−e** option and no **−f**'s, the flag **−e** may be omitted. The **−n** option suppresses the default output.

A script consists of editing commands, one per line, of the following form:

[address [, address]] function [arguments]

In normal operation *sed* cyclically copies a line of input into a *pattern space* (unless there is something left after a 'D' command), applies in sequence all commands whose *addresses* select that pattern space, and at the end of the script copies the pattern space to the standard output (except under **−n**) and deletes the pattern space.

An *address* is either a decimal number that counts input lines cumulatively across files, a '$' that addresses the last line of input, or a context address, '/regular expression/', in the style of *ed*(1) modified thus:

The escape sequence '\n' matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function '!' (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

An argument denoted *text* consists of one or more lines, all but the last of which end with '\' to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an 's' command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument denoted *rfile* or *wfile* must terminate the command line and must be preceded by exactly one blank. Each *wfile* is created before processing begins. There can be at most 10 distinct *wfile* arguments.

(1) a\
text

Append. Place *text* on the output before reading the next input line.

(2) b *label*

Branch to the ':' command bearing the *label*. If *label* is empty, branch to the end of the script.

(2) c\
text

Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place *text* on the output. Start the next cycle.

(2) d Delete the pattern space. Start the next cycle.

(2) D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2) g Replace the contents of the pattern space by the contents of the hold space.

(2) G Append the contents of the hold space to the pattern space.

(2) h Replace the contents of the hold space by the contents of the pattern space.

(2) H Append the contents of the pattern space to the hold space.

(1) i\
text
Insert. Place *text* on the standard output.

(2) n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2) N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2) p Print. Copy the pattern space to the standard output.

(2) P Copy the initial segment of the pattern space through the first newline to the standard output.

(1) q Quit. Branch to the end of the script. Do not start a new cycle.

(2) r *rfile*
Read the contents of *rfile*. Place them on the output before reading the next input line.

(2) s/*regular expression*/*replacement*/*flags*
Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character may be used instead of '/'. For a fuller description see *ed*(1). *Flags* is zero or more of

- g Global. Substitute for all nonoverlapping instances of the *regular expression* rather than just the first one.
- p Print the pattern space if a replacement was made.
- w *wfile* Write. Append the pattern space to *wfile* if a replacement was made.

(2) t *label*
Test. Branch to the ':' command bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a 't'. If *label* is empty, branch to the end of the script.

(2) w *wfile*
Write. Append the pattern space to *wfile*.

(2) x Exchange the contents of the pattern and hold spaces.

(2) y/*string1*/*string2*/
Transform. Replace all occurrences of characters in *string1* with the corresponding character in *string2*. The lengths of *string1* and *string2* must be equal.

(2)! *function*
Don't. Apply the *function* (or group, if *function* is '{') only to lines *not* selected by the address(es).

(0) : *label*
This command does nothing; it bears a *label* for 'b' and 't' commands to branch to.

(1) = Place the current line number on the standard output as a line.

(2) { Execute the following commands through a matching '}' only when the pattern space is selected.

(0) An empty command is ignored.

SEE ALSO

ed(1), grep(1), awk(1)

NAME

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 as ^x, for some x; delete prints as ^?. For full information see *cat*(1), as *see* is a synonym for the **−v** option to *cat.*

SEE ALSO

cat(1)

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**). A word that begins with a # introduces a comment; that word and the rest of the line is ignored.

The first word of a simple-command specifies the name of the command to be executed. Except as specified below the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec*(2)). The *value* of a simple-command is its exit status if it terminates normally or 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of one or more *commands* separated by **|**. The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

A *list* is a sequence of one or more *pipelines* separated by **;**, **&**, **&&** or **||** and optionally terminated by **;** or **&**. **;** and **&** have equal precedence which is lower than that of **&&** and **||**, **&&** and **||** also have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding *pipeline* to be executed without waiting for it to finish. The symbol **&&** (**||**) causes the *list* following to be executed only if the preceding *pipeline* returns a zero (non zero) value. Newlines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a simple-command or one of the following. The value returned by a command is that of the last simple-command executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**

Each time a **for** command is executed *name* is set to the next word in the **for** word list If **in** *word* ... is omitted then **in "$@"** is assumed. Execution ends when there are no more words in the list.

case *word* **in** [*pattern* [**|** *pattern*] ... **)** *list* **;;**] ... **esac**

A **case** command executes the *list* associated with the first pattern that matches *word*. The form of the patterns is the same as that used for file name generation.

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**

The *list* following **if** is executed and if it returns zero the *list* following **then** is executed. Otherwise, the *list* following **elif** is executed and if its value is zero the *list* following **then** is executed. Failing that the **else** *list* is executed.

while *list* [**do** *list*] **done**

A **while** command repeatedly executes the **while** *list* and if its value is zero executes the **do** *list;* otherwise the loop terminates. The value returned by a **while** command is that of the last executed command in the **do** *list*. **until** may be used in place of **while** to negate the loop termination test.

(*list* **)** Execute *list* in a subshell.

{ *list* **}** *list* is simply executed.

The following words are only recognized as the first word of a command and when not quoted.

if then else elif fi case in esac for while until do done { }

Command substitution.
The standard output from a command enclosed in a pair of grave accents (\`\`) may be used as part or all of a word; trailing newlines are removed.

Parameter substitution.
The character **$** is used to introduce substitutable parameters. Positional parameters may be assigned values by **set**. Variables may be set by writing

name = *value* [*name* = *value*] ...

${*parameter*}
: A *parameter* is a sequence of letters, digits or underscores (a *name*), a digit, or any of the characters *** @ # ? − $!**. The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is a digit then it is a positional parameter. If *parameter* is ***** or **@** then all the positional parameters, starting with **$1**, are substituted separated by spaces. **$0** is set from argument zero when the shell is invoked.

${*parameter* −*word*}
: If *parameter* is set then substitute its value; otherwise substitute *word*.

${*parameter* = *word*}
: If *parameter* is not set then set it to *word;* the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${*parameter* ? *word*}
: If *parameter* is set then substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted then a standard message is printed.

${*parameter*+ *word*}
: If *parameter* is set then substitute *word;* otherwise substitute nothing.

In the above *word* is not evaluated unless it is to be used as the substituted string. (So that, for example, echo ${d−\`pwd\`} will only execute *pwd* if *d* is unset.)

The following *parameters* are automatically set by the shell.

#	The number of positional parameters in decimal.
−	Options supplied to the shell on invocation or by **set**.
?	The value returned by the last executed command in decimal.
$	The process number of this shell.
!	The process number of the last background command invoked.

The following *parameters* are used but not set by the shell.

HOME	The default argument (home directory) for the **cd** command.
PATH	The search path for commands (see **execution**).
MAIL	If this variable is set to the name of a mail file then the shell informs the user of the arrival of mail in the specified file.
PS1	Primary prompt string, by default '**$** '.
PS2	Secondary prompt string, by default '**>** '.
IFS	Internal field separators, normally **space**, **tab**, and **newline**.

Blank interpretation.
After parameter and command substitution, any results of substitution are scanned for internal field separator characters (those found in **$IFS**) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments

(those resulting from *parameters* that have no values) are removed.

File name generation.
Following substitution, each command word is scanned for the characters *, ? and [. If one of these characters appears then the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern then the word is left unchanged. The character . at the start of a file name or immediately following a /, and the character /, must be matched explicitly.

- `*` Matches any string, including the null string.
- `?` Matches any single character.
- `[...]` Matches any one of the characters enclosed. A pair of characters separated by − matches any character lexically between the pair.

Quoting.
The following characters have a special meaning to the shell and cause termination of a word unless quoted.

; & ( ) | < > **newline space tab**

A character may be *quoted* by preceding it with a \. **newline** is ignored. All characters enclosed between a pair of quote marks (''), except a single quote, are quoted. Inside double quotes ("") parameter and command substitution occurs and \ quotes the characters \ ` " 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 the occur after the command name. The following prints 'a=b c' and 'c':

```
echo a=b c
set −k
echo a=b c
```

Signals.
The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent. (But see also **trap.**)

Execution.
Each time a command is executed the above substitutions are carried out. Except for the 'special commands' listed below a new process is created and an attempt is made to execute the command via an *exec*(2).

The shell parameter **$PATH** defines the search path for the directory containing the command. Each alternative directory name is separated by a colon (:). The default path is **:/bin:/usr/bin**. If the command name contains a / then the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an *a.out* file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a subshell.

Special commands.
The following commands are executed in the shell process and except where specified no input output redirection is permitted for such commands.

: No effect; the command does nothing.

. *file* Read and execute commands from *file* and return. The search path **$PATH** is used to find the directory containing *file*.

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified then break *n* levels.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified then resume at the *n*-th enclosing loop.

cd [*arg*]
Change the current directory to *arg*. The shell parameter **$HOME** is the default *arg*.

eval [*arg* ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [*arg* ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input output arguments may appear and if no other arguments are given cause the shell input output to be modified.

exit [*n*]
Causes a non interactive shell to exit with the exit status specified by *n*. If *n* is omitted then the exit status is that of the last command executed. (An end of file will also exit from the shell.)

export [*name* ...]
The given names are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given then a list of exportable names is printed.

login [*arg* ...]
Equivalent to 'exec login arg ...'.

newgrp [*arg* ...]
Equivalent to 'exec newgrp arg ...'.

read *name* ...
One line is read from the standard input; successive words of the input are assigned to the variables *name* in order, with leftover words to the last variable. The return code is 0 unless the end-of-file is encountered.

readonly [*name* ...]
The given names are marked readonly and the values of the these names may not be changed by subsequent assignment. If no arguments are given then a list of all readonly names is printed.

set [**−eknptuvx** [*arg* ...]]

- **−e** If non interactive then exit immediately if a command fails.
- **−k** All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- **−n** Read commands but do not execute them.
- **−t** Exit after reading and executing one command.
- **−u** Treat unset variables as an error when substituting.
- **−v** Print shell input lines as they are read.
- **−x** Print commands and their arguments as they are executed.
- **−** Turn off the **−x** and **−v** options.

These flags can also be used upon invocation of the shell. The current set of flags may be found in **$−**.

Remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, etc. If no arguments are given then the values of all names are printed.

shift The positional parameters from **$2**... are renamed **$1**...

times Print the accumulated user and system times for processes run from the shell.

trap [*arg*] [*n*] ...
Arg is a command to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. If *arg* is absent then all trap(s) *n*

are reset to their original values. If *arg* is the null string then this signal is ignored by the shell and by invoked commands. If *n* is 0 then the command *arg* is executed on exit from the shell, otherwise upon receipt of signal *n* as numbered in *signal*(2). *Trap* with no arguments prints a list of commands associated with each signal number.

umask [*nnn*]
The user file creation mask is set to the octal value *nnn* (see *umask*(2)). If *nnn* is omitted, the current value of the mask is printed.

wait [*n*]
Wait for the specified process and report its termination status. If *n* is not given then all currently active child processes are waited for. The return code from this command is that of the process waited for.

Invocation.
If the first character of argument zero is −, commands are read from **$HOME/.profile**, if such a file exists. Commands are then read as described below. The following flags are interpreted by the shell when it is invoked.

−c *string* If the −c flag is present then commands are read from *string*.

−s If the −s flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

−i If the −i flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case the terminate signal SIGTERM (see *signal*(2)) is ignored (so that 'kill 0' does not kill an interactive shell) and the interrupt signal SIGINT is caught and ignored (so that **wait** is interruptable). In all cases SIGQUIT is ignored by the shell.

The remaining flags and arguments are described under the **set** command.

FILES
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO
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 [object ...]

DESCRIPTION

Size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in hex and decimal, of each object-file argument. If no file is specified, **a.out** is used.

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, as in:

```
while true
do
        command
        sleep 37
done
```

SEE ALSO

alarm(2), sleep(3)

BUGS

Time must be less than 2147483647 seconds.

NAME

soelim − eliminate .so's from nroff input

SYNOPSIS

soelim [file ...]

DESCRIPTION

Soelim reads the specified files or the standard input and performs the textual inclusion implied by the *nroff* directives of the form

.so somefile

when they appear at the beginning of input lines. This is useful since programs such as *tbl* do not normally do this; it allows the placement of individual tables in separate files to be run as a part of a large document.

Note that inclusion can be suppressed by using '′' instead of '.', i.e.

′so /usr/lib/tmac.s

A sample usage of *soelim* would be

soelim exum?.n | tbl | nroff −ms | col | lpr

SEE ALSO

colcrt(1), more(1)

AUTHOR

William Joy

BUGS

The format of the source commands must involve no strangeness − exactly one blank must precede and no blanks follow the file name.

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*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr**, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first nonblank in the field; **b** is attached independently to *pos2*. A missing *.n* means .0; a missing **−***pos2* means the end of the line. Under the **−t***x* option, fields are strings separated by *x*; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

m Merge only, the input files are already sorted.

o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

T The next argument is the name of a directory in which temporary files should be made.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

Examples. Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

sort −u +0f +0 list

Print the password file (*passwd*(5)) sorted by user id number (the 3rd colon-separated field).

sort −t: +2n /etc/passwd

Print the first instance of each month in an already sorted file of (month day) entries. The options **−um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

sort −um +0 −1 dates

FILES

/usr/tmp/stm*, /tmp/* first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **−c**.

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 neither occur among nor are derivable (by applying certain inflections, prefixes or suffixes) from words in the spelling list are printed on the standard output. If no files are named, words are collected from the standard input.

Spell ignores most *troff, tbl* and *eqn*(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,* Fowler and the OED to the contrary notwithstanding.

Under the **−x** option, every plausible stem is printed with '=' for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective in respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy−y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by *spell.* Both expect a list of words, one per line, from the standard input. *Spellin* adds the words on the standard input to the preexisting *list* and places a new list on the standard output. If no *list* is specified, the new list is created from scratch. *Spellout* looks up each word in the standard input and prints on the standard output those that are missing from (or present on, with option **−d**) the hash list.

FILES

D=/usr/dict/hlist[ab]: hashed spelling lists, American & British
S=/usr/dict/hstop: hashed stop list
H=/usr/dict/spellhist: history file
/usr/lib/spell
deroff(1), sort(1), tee(1), sed(1)

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

spline — interpolate smooth curve

SYNOPSIS

spline [option] ...

DESCRIPTION

Spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output (R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *graph*(1).

The following options are recognized, each as a separate argument.

−a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

−k The constant k used in the boundary value computation

$$y_0'' = ky_1'', \quad y_n'' = ky_{n-1}''$$

is set by the next argument. By default $k = 0$.

−n Space output points so that approximately n intervals occur between the lower and upper x limits. (Default $n = 100$.)

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

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

SEE ALSO

graph(1)

DIAGNOSTICS

When data is not strictly monotone in x, *spline* reproduces the input without interpolating extra points.

BUGS

A limit of 1000 input points is enforced silently.

NAME

split − split a file into pieces

SYNOPSIS

split [−*n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, **x** is default.

If no input file is given, or if − is given in its stead, then the standard input file is used.

NAME

strings − find the printable strings in a 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.

SEE ALSO

od(1)

AUTHOR

Bill Joy

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*.

FILES

/tmp/stm? temporary file

SEE ALSO

ld(1)

NAME

struct − structure Fortran programs

SYNOPSIS

struct [option] ... file

DESCRIPTION

Struct translates the Fortran program specified by *file* (standard input default) into a Ratfor program. Wherever possible, Ratfor control constructs replace the original Fortran. Statement numbers appear only where still necessary. Cosmetic changes are made, including changing Hollerith strings into quoted strings and relational operators into symbols (.e.g. ".GT." into ">"). The output is appropriately indented.

The following options may occur in any order.

−s Input is accepted in standard format, i.e. comments are specified by a c, C, or * in column 1, and continuation lines are specified by a nonzero, nonblank character in column 6. Normally input is in the form accepted by *f77*(1)

−i Do not turn computed goto statements into switches. (Ratfor does not turn switches back into computed goto statements.)

−a Turn sequences of else ifs into a non-Ratfor switch of the form

```
switch
        {   case pred1: code
            case pred2: code
            case pred3: code
            default: code
        }
```

The case predicates are tested in order; the code appropriate to only one case is executed. This generalized form of switch statement does not occur in Ratfor.

−b Generate goto's instead of multilevel break statements.

−n Generate goto's instead of multilevel next statements.

−t*n* Make the nonzero integer *n* the lowest valued label in the output program (default 10).

−c*n* Increment successive labels in the output program by the nonzero integer *n* (default 1).

−e*n* If *n* is 0 (default), place code within a loop only if it can lead to an iteration of the loop. If *n* is nonzero, admit a small code segments to a loop if otherwise the loop would have exits to several places including the segment, and the segment can be reached only from the loop. 'Small' is close to, but not equal to, the number of statements in the code segment. Values of n under 10 are suggested.

FILES

/tmp/struct*
/usr/lib/struct/*

SEE ALSO

f77(1)

BUGS

Struct knows Fortran 66 syntax, but not full Fortran 77.
If an input Fortran program contains identifiers which are reserved words in Ratfor, the structured version of the program will not be a valid Ratfor program.
The labels generated cannot go above 32767.
If you get a goto without a target, try **−e** .

NAME

stty — set terminal options

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output terminal, placing its output on the diagnostic output. With no argument, it reports the speed of the terminal and the settings of the options which are different from their defaults. With the argument "all", all normally used option settings are reported. With the argument "everything", everything *stty* knows about is printed. The option strings are selected from the following set:

even allow even parity input

−even disallow even parity input

odd allow odd parity input

−odd disallow odd parity input

raw raw mode input (**no** input processing (erase, kill, interrupt, ...); 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 processing, but all other processing (interrupt, suspend, ...) is performed

−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

tandem enable flow control, so that the system sends out the stop character when its internal queue is in danger of overflowing on input, and sends the start character when it is ready to accept further input

−tandem disable flow control

−tabs replace tabs by spaces when printing

tabs preserve tabs

ek set erase and kill characters to # and @

For the following commands which take a character argument *c*, you may also specify *c* as the "u" or "undef", to set the value to be undefined. A value of "^x", a 2 character sequence, is also interpreted as a control character, with "^?" representing delete.

erase *c* set erase character to *c* (default '#', but often reset to ^H.)

kill *c* set kill character to *c* (default '@', but often reset to ^U.)

intr *c* set interrupt character to *c* (default DEL or ^? (delete), but often reset to ^C.)

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

bs0 bs1 select style of delay for backspace

tty33 set all modes suitable for the Teletype Corporation Model 33 terminal.

tty37 set all modes suitable for the Teletype Corporation Model 37 terminal.

vt05 set all modes suitable for Digital Equipment Corp. VT05 terminal

dec set all modes suitable for Digital Equipment Corp. operating systems users; (erase, kill, and interrupt characters to ^?, ^U, and ^C, decctlq and "newcrt".)

tn300 set all modes suitable for a General Electric TermiNet 300

ti700 set all modes suitable for Texas Instruments 700 series terminal

tek set all modes suitable for Tektronix 4014 terminal

0 hang up phone line immediately

50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb
Set terminal baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

A teletype driver which supports the job control processing of *csh*(1) as introduced in *newcsh*(1) and more functionality that the basic driver is introduced in *newtty*(4) and fully described in *tty*(4). The following options apply only to it.

new Use new driver (switching flushes typeahead).

crt Set options for a CRT (crtbs, ctlecho and, if >= 1200 baud, crterase and crtkill.)

crtbs Echo backspaces on erase characters.

prterase For printing terminal echo erased characters backwards within "\" and "/".

crterase Wipe out erased characters with "backspace-space-backspace."

−crterase Leave erased characters visible; just backspace.

crtkill Wipe out input on like kill ala **crterase.**

−crtkill Just echo line kill character and a newline on line kill.

ctlecho Echo control characters as "^*x*" (and delete as "^?".) Print two backspaces following the EOT character (control D).

−ctlecho Control characters echo as themselves; in cooked mode EOT (control-D) is not echoed.

decctlq After output is suspended (normally by ^S), only a start character (normally ^Q) will restart it. This is compatible with DEC's vendor supplied systems.

−decctlq After output is suspended, any character typed will restart it; the start character will restart output without providing any input. (This is the default.)

tostop Background jobs stop if they attempt terminal output.

−tostop Output from background jobs to the terminal is allowed.

tilde Convert "~" to "`" on output (for Hazeltine terminals).

−tilde Leave poor "~" alone.

flusho Output is being discarded usually because user hit control O (internal state bit).

−flusho Output is not being discarded.

pendin Input is pending after a switch from cbreak to cooked and will be re-input when a read becomes pending or more input arrives (internal state bit).

−pendin Input is not pending.

intrup Send a signal (SIGTINT) to the terminal control process group whenever an input record (line in cooked mode, character in cbreak or raw mode) is available for reading.

−intrup Don't send input available interrupts.

mdmbuf Start/stop output on carrier transitions (not implemented).

−mdmbuf
Return error if write attempted after carrier drops.

litout	Send output characters without any processing.
−litout	Do normal output processing, inserting delays, etc.
nohang	Don't send hangup signal if carrier drops.
−nohang	Send hangup signal to control process group when carrier drops.
etxack	Diablo style etx/ack handshaking (not implemented).

The following special characters are applicable only to the new teletype driver and are not normally changed.

susp *c*	set suspend process character to *c* (default control Z.)
dsusp *c*	set delayed suspend process character to *c* (default control Y.)
rprnt *c*	set reprint line character to *c* (default control R.)
flush *c*	set flush output character to *c* (default control O.)
werase *c*	set word erase character to *c* (default control W.)
lnext *c*	set literal next character to *c* (default control V.)

SEE ALSO

ioctl(2), tabs(1), tset(1), newtty(4), tty(4)

NAME

style − analyze surface characteristics of a document

SYNOPSIS

style [**−ml**] [**−mm**] [**−a**] [**−e**] [**−l** num] [**−r** num] [**−p**] [**−P**] file ...

DESCRIPTION

Style analyzes the surface characteristics of the writing style of a document. It reports on readability, sentence length and structure, word length and usage, verb type, and sentence openers. Because *style* runs *deroff* before looking at the text, formatting header files should be included as part of the input. The default macro package **−ms** may be overridden with the flag **−mm.** The flag **−ml**, which causes **deroff** to skip lists, should be used if the document contains many lists of non-sentences. The other options are used to locate sentences with certain characteristics.

−a print all sentences with their length and readability index.

−e print all sentences that begin with an expletive.

−p print all sentences that contain a passive verb.

−l*num* print all sentences longer than *num*.

−r*num* print all sentences whose readability index is greater than *num*.

−P print parts of speech of the words in the document.

SEE ALSO

deroff(1), diction(1)

BUGS

Use of non-standard formatting macros may cause incorrect sentence breaks.

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) 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.

If no *userid* is specified, 'root' is assumed. To remind the super-user of his responsibilities, the Shell substitutes '#' for its usual prompt.

SEE ALSO

sh(1)

BUGS

Local administrative rules cause restrictions to be placed on who can *su* to 'root', even with the root password. These rules vary from site to site.

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. It is typically used to look for bad spots, or to validate a file communicated over some transmission line.

SEE ALSO

wc(1)

DIAGNOSTICS

'Read error' is indistinuishable from end of file on most devices; check the block count.

NAME

symorder — rearrange name list

SYNOPSIS

symorder orderlist symbolfile

DESCRIPTION

Orderlist is a file containing symbols to be found in symbolfile, 1 symbol per line.

Symbolfile is updated in place to put the requested symbols first in the symbol table, in the order specified. This is done by swapping the old symbols in the required spots with the new ones. If all of the order symbols are not found, an error is generated.

This program was specifically designed to cut down on the overhead of getting symbols from /vmunix.

SEE ALSO

nlist(3)

NAME

tabs − set terminal tabs

SYNOPSIS

tabs [**−n**] [terminal]

DESCRIPTION

Tabs sets the tabs on a variety of terminals. Various terminal names given in *term*(7) are recognized; the default is, however, suitable for most 300 baud terminals. If the **−n** flag is present then the left margin is not indented as is normal.

SEE ALSO

stty(1), term(7)

BUGS

It's much better to use *tset*(1).

NAME

tail – deliver the last part of a file

SYNOPSIS

tail [±number[**lbc**][**fr**]] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance *+number* from the beginning, or *−number* from the end of the input. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b** or **c**. When no units are specified, counting is by lines.

Specifying **r** causes tail to print lines from the end of the file in reverse order. The default for **r** is to print the entire file this way. Specifying **f** causes *tail* to not quit at end of file, but rather wait and try to read repeatedly in hopes that the file will grow.

SEE ALSO

dd(1)

BUGS

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length.

Various kinds of anomalous behavior may happen with character special files.

NAME

tar − tape archiver

SYNOPSIS

tar [key] [name ...]

DESCRIPTION

Tar saves and restores files on 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 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**.

o On output, tar normally places information specifying owner and modes of directories in the archive. Former versions of tar, when encountering this information will give error message of the form

"<name>/: cannot create".

This option will suppress the directory information.

p This option says to restore files to their original modes, ignoring the present umask(2). Setuid and sticky information will also be restored to the super-user.

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 with the command

cd fromdir; tar cf - . | (cd todir; tar xf -)

b causes *tar* to use the next argument as the blocking factor for tape records. The default is 20 (the maximum). This option should only be used with raw magnetic tape archives (See **f** above). The block size is determined automatically when reading tapes (key letters 'x' and 't').

l tells *tar* to complain if it cannot resolve all of the links to the files dumped. If this is not specified, no error messages are printed.

m tells *tar* to not restore the modification times. The mod time will be the time of extraction.

Previous restrictions dealing with *tar*'s inability to properly handle blocked archives have been lifted.

FILES

/dev/rmt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if enough memory is not available to hold the link tables.

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 reformatted. Details are given in the reference manual.

As an example, letting \t represent a tab (which should be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\t1644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields

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

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

SEE ALSO

troff(1), eqn(1)
M. E. Lesk, *TBL*.

NAME

tc − photypesetter simulator

SYNOPSIS

tc [**−t**] [**−sN**] [**−pL**] [file]

DESCRIPTION

Tc interprets its input (standard input default) as device codes for a Graphic Systems phototypesetter (cat). The standard output of *tc* is intended for a Tektronix 4015 (a 4014 teminal with ASCII and APL character sets). The sixteen typesetter sizes are mapped into the 4014's four sizes; the entire TROFF character set is drawn using the 4014's character generator, using overstruck combinations where necessary. Typical usage:

troff −t file | tc

At the end of each page *tc* waits for a newline (empty line) from the keyboard before continuing on to the next page. In this wait state, the command e will suppress the screen erase before the next page; sN will cause the next N pages to be skipped; and !line will send line to the shell.

The command line options are:

−t Don't wait between pages; for directing output into a file.

−sN Skip the first N pages.

−pL Set page length to L. L may include the scale factors **p** (points), **i** (inches), **c** (centimeters), and **P** (picas); default is picas.

'−*l w*' Multiply the default aspect ratio, 1.5, of a displayed page by *l/w*.

SEE ALSO

troff(1), plot(1)

BUGS

Font distinctions are lost.
tc's character set is limited to ASCII in just one size.
The aspect ratio option is unbelievable.

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.

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 time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

On a PDP-11, the execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on the diagnostic output stream.

BUGS

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

Time is a built-in command to *csh*(1), with a much different syntax. This command is available as "/bin/time" to *csh* users.

NAME

tk − paginator for the Tektronix 4014

SYNOPSIS

tk [**−t**] [**−***N*] [**−p***L*] [file]

DESCRIPTION

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

The command line options are:

−t Don't wait between pages; for directing output into a file.

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

−p*L* Set page length to *L* lines.

SEE ALSO

pr(1)

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.

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:

r The named files are written on the tape. If files with the same names already exist, they are replaced. 'Same' is determined by string comparison, so './abc' can never be the same as '/usr/dmr/abc' even if '/usr/dmr' is the current directory. If no file argument is given, '.' is the default.

u updates the tape. **u** is like **r**, but a file is replaced only if its modification date is later than the date stored on the tape; that is to say, if it has changed since it was dumped. **u** is the default command if none is given.

d deletes the named files from the tape. At least one name argument must be given. This function is not permitted on magtapes.

x extracts the named files from the tape to the file system. The owner and mode are restored. If no file argument is given, the entire contents of the tape are extracted.

t lists the names of the specified files. If no file argument is given, the entire contents of the tape is listed.

The following characters may be used in addition to the letter which selects the function desired.

m Specifies magtape as opposed to DECtape.

0,...,7 This modifier selects the drive on which the tape is mounted. For DECtape, **x** is default; for magtape '0' is the default.

v Normally *tp* does its work silently. The **v** (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.

c means a fresh dump is being created; the tape directory is 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 currently acts like **m**; *i.e.* **r** implies **c**, and neither **d** nor **u** are permitted.

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/rmt?

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.

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect '\' from the Shell. 012 is the ASCII code for newline.

```
tr −cs A−Za−z '\012' <file1 >file2
```

SEE ALSO

ed(1), ascii(7)

BUGS

Won't handle ASCII NUL in *string1* or *string2;* always deletes NUL from input.

NAME

trman — translate version 6 manual macros to version 7 macros

SYNOPSIS

trman [file]

DESCRIPTION

Trman reads the input file, which should be nroff/troff input and attempts to translate the version 6 manual sections therein to version 7 format. It is largely successful, but seems to have trouble with indented paragraphs and complicated font control. You should expect to have to fix up long sections by hand somewhat.

SEE ALSO

man(7)

BUGS

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* 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.

Nroff only

−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.

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 elasped 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*
vtroff(1), eqn(1), tbl(1), pti(1), ms(7), me(7), man(7), soelim(1)
col(1), tk(1) (*nroff* only)
tc(1) (*troff* only)

NAME

true, false — provide truth values

SYNOPSIS

true

false

DESCRIPTION

True does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1) such as:

```
while true
do
        command
done
```

SEE ALSO

sh(1)

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

tset − set terminal modes

SYNOPSIS

tset [options] [**−m** [*ident*][*test baudrate*]:*type* ...] [type]

DESCRIPTION

Tset causes terminal dependent processing such as setting erase and kill characters, setting or resetting delays, and the like. It 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 on terminals which are not hardwired. 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):

    set noglob; eval `tset −s *options...*`

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).

The 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.

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
tsort — topological sort

SYNOPSIS
tsort [file]

DESCRIPTION
Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO
lorder(1)

DIAGNOSTICS
Odd data: there is an odd number of fields in the input file.

BUGS
Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME

tty — get terminal name

SYNOPSIS

tty

DESCRIPTION

Tty prints the pathname of the user's terminal.

DIAGNOSTICS

'not a tty' if the standard input file is not a terminal.

NAME

ul − do underlining

SYNOPSIS

ul [**−i**] [**−t** *terminal*] [*name* ...]

DESCRIPTION

Ul reads the named files (or standard input if none are given) and translates occurances of underscores to the sequence which indicates underlining for the terminal in use, as specified by the environment variable TERM. The **−t** option overrides the terminal kind specified in the environment. The file */etc/termcap* is read to determine the appropriate sequences for underlining. If the terminal is incapable of underlining, but is capable of a standout mode then that is used instead. If the terminal can overstrike, or handles underlining automatically, *ul* degenerates to *cat*(1). If the terminal cannot underline, underlining is ignored.

The **−i** option causes *ul* to indicate underlining onto by a separate line containing appropriate dashes '−'; this is useful when you want to look at the underlining which is present in an *nroff* output stream on a crt-terminal.

SEE ALSO

man(1), nroff(1), colcrt(1)

AUTHOR

Mark Horton wrote *ul;* the **−i** option was originally a option of the editor *ex,* then a *iul* command.

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

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/units'.

FILES

/usr/lib/units

BUGS

Don't base your financial plans on the currency conversions.

NAME

uptime – show how long system has been up

SYNOPSIS

uptime

DESCRIPTION

Uptime prints the current time, the length of time the system has been up, and the average number of jobs in the run queue over the last 1, 5 and 15 minutes. It is, essentially, the first line of a *w* (1) command.

FILES

/vmunix system name list

SEE ALSO

w(1)

NAME

users — compact list of users who are on the system

SYNOPSIS

users

DESCRIPTION

Users lists the login names of the users currently on the system in a compact, one-line format.

FILES

/etc/utmp

SEE ALSO

finger(1), who(1)

BUGS

NAME

uuclean — uucp spool directory clean-up

SYNOPSIS

uuclean [option] ...

DESCRIPTION

Uuclean will scan the spool directory for files with the specified prefix and delete all those which are older than the specified number of hours.

The following options are available.

−p*pre* Scan for files with *pre* as the file prefix. Up to 10 **−p** arguments may be specified. A **−p** without any *pre* following will cause all files older than the specified time to be deleted.

−n*time* Files whose age is more than *time* hours will be deleted if the prefix test is satisfied. (default time is 72 hours)

−m Send mail to the owner of the file when it is deleted.

This program will typically be started by *cron*(8).

FILES

/usr/lib/uucp	directory with commands used by uuclean internally
/usr/lib/uucp/spool	spool directory

SEE ALSO

uucp(1C), uux(1C)

NAME

uucp, uulog − unix to unix copy

SYNOPSIS

uucp [option] ... source-file ... destination-file

uulog [option] ...

DESCRIPTION

Uucp copies files named by the source-file arguments to the destination-file argument. A file name may be a path name on your machine, or may have the form

system-name!pathname

where 'system-name' is taken from a list of system names which *uucp* knows about. Shell metacharacters ?*[] appearing in the pathname part will be expanded on the appropriate system.

Pathnames may be one of

(1) a full pathname;

(2) a pathname preceded by ~*user*; where *user* is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

If the result is an erroneous pathname for the remote system the copy will fail. If the destination-file is a directory, the last part of the source-file name is used.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*.

−d Make all necessary directories for the file copy.

−c Use the source file when copying out rather than copying the file to the spool directory.

−m Send mail to the requester when the copy is complete.

Uulog maintains a summary log of *uucp* and *uux*(1) transactions in the file '/usr/spool/uucp/LOGFILE' by gathering information from partial log files named '/usr/spool/uucp/LOG.*.?'. It removes the partial log files.

The options cause *uulog* to print logging information:

−s*sys* Print information about work involving system *sys*.

−u*user*
Print information about work done for the specified *user*.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/* - other data and program files

SEE ALSO

uux(1), mail(1)
D. A. Nowitz, *Uucp Implementation Description*

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will very likely not be able to fetch files by pathname; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary pathnames.

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

uudiff — directory comparison between machines

SYNOPSIS

uudiff [−d] local-name remote-name

DESCRIPTION

Uudiff compares the files in the directory *local-name* and the directory *remote-name*, (where *remote-name* may be of the form *system-name!directory-name* and *system-name* is a *uucp* Unix name). It reports (via mail) which files are added, deleted, or changed, and provides a *diff(1)* of altered printable files.

If a part of *remote-name* is omitted (either the system or the directory) the corresponding part of *local-name* is used. If *local-name* is a file, rather than a directory, *remote-name* is also assumed to be a file and the program degenerates into *diff(1)*.

The option **−d** does not diff altered files; only the summary by file names is prepared.

FILES

Lots. Files zz[abcdeglmn]????? are used for scratch space; files brought back from the remote machine for *diffing* are stored (and left around) as *name.?????* and the final report is left in *uudiff.?????* (the exact name is reported by mail).

SEE ALSO

diff(1), uucp(1)

DIAGNOSTICS

Almost none. Anything more serious than misspelling *local-name* causes unpredictable and obscure results.

BUGS

In addition to the standard *uucp* requirements a hook is needed at the remote system, and at present is only installed on the systems "research" and "inter".
This program is written in shell and should be translated to C so it could give diagnostics.
Even if "remote-system" is the local system, uudiff is subject to delays in uucp traffic.
It should probably write the standard output, instead of insisting on going into the background.
Since checksums are used there is a probability of 1 in 2**32 of missing differences between files.
The ~userid syntax is not recognized.

NAME

uuencode,uudecode — encode/decode a binary file for tranmission via mail

SYNOPSIS

uuencode [source] remotedest | **mail** sys1!sys2!..!decode
uudecode [file]

DESCRIPTION

Uuencode and *uudecode* are used to send a binary file via uucp (or other) mail. This combination can be used over indirect mail links even when *uusend(1)* is not available.

Uuencode takes the named source file (default standard input) and produces an encoded version on the standard output. The encoding uses only printing ASCII characters, and includes the mode of the file and the *remotedest* for recreation on the remote system.

Uudecode reads an encoded file, strips off any leading and trailing lines added by mailers, and recreates the original file with the specified mode and name.

The intent is that all mail to the user "decode" should be filtered through the uudecode program. This way the file is created automatically without human intervention. This is possible on the uucp network by either using *delivermail* or by making *rmail* be a link to *Mail* instead of *mail.* In each case, an alias must be created in a master file to get the automatic invocation of uudecode.

If these facilities are not available, the file can be sent to a user on the remote machine who can uudecode it manually.

The encode file has an ordinary text form and can be edited by any text editor to change the mode or remote name.

SEE ALSO

uuencode(5), uusend(1), uucp(1), uux(1), mail(1)

AUTHOR

Mark Horton

BUGS

The file is expanded by 35% (3 bytes become 4 plus control information) causing it to take longer to transmit.

The user on the remote system who is invoking *uudecode* (often *uucp)* must have write permission on the specified file.

NAME

uusend — send a file to a remote host

SYNOPSIS

uusend [**−m** mode] sourcefile sys1!sys2!..!remotefile

DESCRIPTION

Uusend sends a file to a given location on a remote system. The system need not be directly connected to the local system, but a chain of *uucp(1)* links needs to connect the two systems.

If the **−m** option is specified, the mode of the file on the remote end will be taken from the octal number given. Otherwise, the mode of the input file will be used.

The sourcefile can be "−", meaning to use the standard input. Both of these options are primarily intended for internal use of uusend.

The remotefile can include the ~userid syntax.

DIAGNOSTICS

If anything goes wrong any further away than the first system down the line, you will never hear about it.

SEE ALSO

uux(1), uucp(1), uuencode(1)

AUTHOR

Mark Horton

BUGS

This command shouldn't exist, since *uucp* should handle it.

All systems along the line must have the *uusend* command available and allow remote execution of it.

Some uucp systems have a bug where binary files cannot be the input to a uux command. If this bug exists in any system along the line, the file will show up severly munged.

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/uucp/spool - spool directory
/usr/uucp/* - other data and programs

SEE ALSO

uucp(1)
D. A. Nowitz, *Uucp implementation description*

WARNING

An installation may, and for security reasons generally will, limit the list of commands executable on behalf of an incoming request from *uux*. Typically, a restricted site will permit little other than the receipt of mail via *uux*.

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

vfontinfo — inspect and print out information about unix fonts

SYNOPSIS

vfontinfo [**−v**] fontname [characters]

DESCRIPTION

Vfontinfo allows you to examine a font in the unix format. It prints out all the information in the font header and information about every non-null (width > 0) glyph. This can be used to make sure the font is consistent with the format.

The *fontname* argument is the name of the font you wish to inspect. It writes to standard output. If it can't find the file in your working directory, it looks in **/usr/lib/vfont** (the place most of the fonts are kept).

The *characters*, if given, specify certain characters to show. If omitted, the entire font is shown.

If the **−v** (verbose) flag is used, the bits of the glyph itself are shown as an array of X's and spaces, in addition to the header information.

SEE ALSO

vpr(1), vfont(5)
The Berkeley Font Catalog

AUTHORS

Mark Horton
Andy Hertzfeld

NAME

vgrind − grind nice listings of programs

SYNOPSIS

vgrind [**−t**] [**−n**] [**−x**] [**−W**] [**−c**] [**−m**] [**−p**] [**−i**] [**−sn**] [**−h** header] name ...

DESCRIPTION

Vgrind formats the C, MODEL, or PASCAL programs which are arguments in a nice style using *troff* (1). Comments are placed in italics, keywords in bold face, and the name of the current function is listed down the margin of each page as it is encountered.

The **−W** option sends the output to a 4 page wide raster plotter; normally, the output is printed on a narrow plotter.

The **−c**, **−m**, **−p**, and **−i** options select C, MODEL, PASCAL or ISP respectively as the language of the input files. The **−c** switch is the default and need not be specified.

Font size may be specified using the **−s** switch. The argument **n** is the point size (same as the argument of a .ps troff command).

The **−h** option specifies a header to be placed at the top of each page. If the program is not source, but should be framed like the output of *vgrind*, the **−n** option should be specified.

The **−t** option is analogous to the **−t** option of *troff*(1) placing typesetter codes on the standard output.

To create an index, it is only necessary to create an empty file *index* in the current directory. As you run *vgrind*, the index will be automatically kept up to date. The index of function definitions can then be run off via giving *vgrind* the **−x** option and the file *index* as argument.

FILES

index	file where source for index is created
/usr/lib/tmac/tmac.vgrind	macro package
/usr/lib/vfontedpr	preprocessor

AUTHOR

William Joy

SEE ALSO

vtroff(1)

BUGS

Vfontedpr assumes that a certain programming style is followed:

For C - functions begin with the name of the function in column one, and lines defining functions end with a). The function name is followed immediately by a "(" with no intervening space.

For PASCAL - function names need to appear on the same line as the keywords *function* and *procedure*. The keyword *end* at the beginning of a line is interpreted as the end of the current function.

For MODEL - function names need to appear on the same line as the keywords *is beginproc*.

If these conventions are not followed, the indexing and marginal function name comment mechanisms will fail.

More generally, arbitrary formatting styles for programs mostly look bad. The use of spaces to align source code fails miserably; if you plan to *vgrind* your program you should use tabs. This is somewhat inevitable since the font used by *vgrind* is variable width.

Should be able to be used as a preprocessor

The mechanism of ctags in recognizing functions should be used here.

NAME

vi — screen oriented (visual) display editor based on ex

SYNOPSIS

vi [**−t** tag] [**−r**] [**+***command*] [**−l**] [**−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.

The *Vi Quick Reference* card and the *Introduction to Display Editing with Vi* provide full details on using *vi.*

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

vmstat − report virtual memory statistics

SYNOPSIS

vmstat [**−fs**] [interval [count]]

DESCRIPTION

Vmstat delves into the system and normally reports certain statistics kept about process, virtual memory, disk, trap and cpu activity. If given a **−f** argument, it instead reports on the number of *forks* and *vforks* since system startup and the number of pages of virtual memory involved in each kind of fork. If given a **−s** argument, it instead prints the contents of the *sum* structure, giving the total number of several kinds of paging related events which have occurred since boot.

If none of these options are given, *vmstat* will report in a (usually) iterative fashion on the virtual memory activity in the system. In this case, the optional *interval* argument causes *vmstat* to report once each *interval* seconds; "vmstat 5" will print what the system is doing every five seconds; this is a good choice of printing interval since this is how often some of the statistics are sampled in the system; others vary every second, running the output for a while will make it apparent which are recomputed every second. If a *count* is given, the statistics are repeated *count* times. The format fields are:

Procs: information about numbers of processes in various states.

r	in run queue
b	blocked for resources (i/o, paging, etc.)
w	runnable or short sleeper (< 20 secs) but swapped

Memory: information about the usage of virtual and real memory. Virtual pages are considered active if they belong to processes which are running or have run in the last 20 seconds. A "page" here is 1024 bytes.

avm	active virtual pages
fre	size of the free list

Page: information about page faults and paging activity. These are averaged each five seconds, and given in units per second.

re	page reclaims (simulating reference bits)
pi	pages paged in
po	pages paged out
fr	pages freed per second
de	anticipated short term memory shortfall
sr	pages scanned by clock algorithm, per-second

up/hp/rk: Disk operations per second (this field is system dependent). Typically paging will be split across several of the available drives. The number under each of these is the unit number.

Faults: trap/interrupt rate averages per second over last 5 seconds.

in	(non clock) device interrupts per second
sy	system calls per second
cs	cpu context switch rate (switches/sec)

Cpu: breakdown of percentage usage of CPU time

us	user time for normal and low priority processes
sy	system time
id	cpu idle

FILES

/dev/kmem, /vmunix

SEE ALSO

The sections starting with "Interpreting system activity" in *Setting up 4.1bsd* by W. Joy.

AUTHORS

William Joy and Ozalp Babaoglu

BUGS

There should be a screen oriented program which combines *vmstat* and *ps*(1) in real time as well as reporting on other system activity.

NAME

vpr, vprm, vpq, vprint — raster printer/plotter spooler

SYNOPSIS

vpr [**−W**] [**−l**] [**−v**] [**−t** [**−1234** font]] [**−w**] [**−w***width*] [**−m**] [name ...]
vprm [id ...] [filename ...] [owner ...]
vpq
vprint [**−W**] file ...

DESCRIPTION

Vpr causes the named files to be queued for printing or typeset simulation on one of the available raster printer/plotters. If no files are named, the standard input is read. By default the input is assumed to be line printer-like text. For very wide plotters, the input is run through the filter */usr/lib/sidebyside* giving it an argument of **−w106** which arranges it four pages adjacent with 90 column lines (the rest is for the left margin). Since there are 8 lines per inch in the default printer font, *vpr* thus produces 86 lines per page (the top and bottom lines are left blank).

The following options are available:

−l Print the input in a more literal manner. Page breaks are not inserted, and most control characters (except format effectors: \n, \f, etc.) are printed (many control characters print special graphics not in the ASCII character set.) Tab and underline processing is still done. If this option is not given, control characters which are not format effectors are ignored, and page breaks are inserted after an appropriate number of lines have been printed on a page.

−W Queues files for printing on a wide output device, if available. Normally, files are queued for printing on a narrow output device.

−1234 Specifies a font to be mounted on font position *i*. The daemon will construct a *.railmag* file referencing */usr/lib/vfont/name.size*.

−m Report by *mail*(1) when printing is complete.

−w (Applicable only to wide output devices.) Do not run the input through sidebyside. Such processing has been done already, or full (440 character) printer width is desired.

−w*width* Use width *width* rather than 90 for *sidebyside*.

−v Use the filter */usr/lib/vrast* to convert the vectors to raster. The named files must be a parameter and vector file (in that order) created by *versaplot*(3x) routines.

−t Use the filter */usr/lib/vcat* to typeset the input on the printer/plotter. The input must have been generated by *troff*(1) run with the **−t** option. **This is not normally run directly** to wide output devices, since it is wasteful to run only one page across. The program *vtroff*(1) is normally used and arranges, using *vsort*(1) for printing to occur four pages across, conserving paper.

Vprm removes entries from the raster device queues. The id, filename or owner should be that reported by *vpq*. All apropriate files will be removed. Both queues are always searched. The id of each file removed from the queue will be printed.

Vpq prints the queues. Each entry in the queue is printed showing the owner of the queue entry, an identification number, the size of the entry in characters, and the file which is to be printed. The *id* is useful for removing a specific entry from the printer queue using *vprm*

Vprint is a shell script which *pr's* a copy of each named file on one of the electrostatic printer/plotters. The files are normally printed on a narrow device; **−W** option causes them to be printed on a wide device.

FILES

/usr/spool/v?d/*	device spool areas
/usr/lib/v?d	daemons
/usr/lib/vpd	Versatec daemon
/usr/lib/vpf	filter for printer simulation
/usr/lib/*vcat	filter for typeset simulation
/usr/lib/vrast	filter for versaplot
/usr/lib/sidebyside	filter for wide output

SEE ALSO

troff(1), vfont(5), vp(4), pti(1), vtroff(1), versaplot(3x)

BUGS

You can't run bit maps in a queued fashion to the plotters. This is because the volume of the data (more than 1 Megabyte per vertical foot) is unwieldy. Instead you must follow the instructions in *vp*(4) and run your program when the plotter is idle, or run it in the background and have it wait for the device to become idle.

Queued jobs print in directory (seemingly random) order. The plotters are fast enough that this is not a real problem.

The 1's (one's) and l's (lower-case el's) in a Benson-Varian's standard character set look very similar; caution is advised.

Vprm should have options allowing just one of the queues to be searched.

A versatec's hardware character set is rather ugly. *Vprint* should use one of the constant width fonts to produce prettier listings.

NAME

vtroff − troff to a raster plotter

SYNOPSIS

vtroff [**−w**] [**−F** majorfont] [**−123** minorfont] [**−l***length*] [**−x**] troff arguments

DESCRIPTION

Vtroff runs *troff*(1) sending its output through various programs to produce typeset output on a raster plotter such as a Benson-Varian or or a Versatec. The −W option specifies that a wide output device be used; the default is to use a narrow device. The −l (lower case l) option causes the output to be split onto successive pages every *length* inches rather than the default 11".

The default font is a Hershey font. If some other font is desired you can give a −F argument and then the font name. This will place normal, italic and bold versions of the font on positions 1, 2, and 3. To place a font only on a single position, you can give an argument of the form −*n* and the minor font name. A **.r** will be added to the minor font name if needed. Thus "vtroff −ms paper" will set a paper in the Hershey font, while "vtroff −F nonie −ms paper" will set the paper in the (sans serif) nonie font. The **−x** option asks for exact simulation of photo-typesetter output. (I.e. using the width tables for the C.A.T. photo-typesetter)

FILES

/usr/lib/tmac/tmac.vcat	default font mounts and bug fixes
/usr/lib/fontinfo/*	fixes for other fonts
/usr/lib/vfont	directory containing fonts

SEE ALSO

troff(1), nettroff(1), vfont(5), vpr(1)

BUGS

Since some macro packages work correctly only if the fonts named R, I, B, and S are mounted, and since the Versatec fonts have different widths for individual characters than the fonts found on the typesetter, the following dodge was necessary: If you don't use the ".fp" troff directive then you get the widths of the standard typesetter fonts suitable for shipping the output of troff over the network to the computer center A machine for phototypesetting. If, however, you remount the R, I, B and S fonts, then you get the width tables for the Versatec.

NAME

w − who is on and what they are doing

SYNOPSIS

w [**−h**] [**−s**] [user]

DESCRIPTION

W prints a summary of the current activity on the system, including what each user is doing. The heading line shows the current time of day, how long the system has been up, the number of users logged into the system, and the load averages. The load average numbers give the number of jobs in the run queue averaged over 1, 5 and 15 minutes.

The fields output are: the users login name, the name of the tty the user is on, the time of day the user logged on, the number of minutes since the user last typed anything, the CPU time used by all processes and their children on that terminal, the CPU time used by the currently active processes, the name and arguments of the current process.

The **−h** flag suppresses the heading. The **−s** flag asks for a short form of output. In the short form, the tty is abbreviated, the login time and cpu times are left off, as are the arguments to commands. **−l** gives the long output, which is the default.

If a *user* name is included, the output will be restricted to that user.

FILES

/etc/utmp
/dev/kmem
/dev/drum

SEE ALSO

who(1), finger(1), ps(1)

AUTHOR

Mark Horton

BUGS

The notion of the "current process" is muddy. The current algorithm is "the highest numbered process on the terminal that is not ignoring interrupts, or, if there is none, the highest numbered process on the terminal". This fails, for example, in critical sections of programs like the shell and editor, or when faulty programs running in the background fork and fail to ignore interrupts. (In cases where no process can be found, *w* prints "−".)

The CPU time is only an estimate, in particular, if someone leaves a background process running after logging out, the person currently on that terminal is "charged" with the time.

Background processes are not shown, even though they account for much of the load on the system.

Sometimes processes, typically those in the background, are printed with null or garbaged arguments. In these cases, the name of the command is printed in parentheses.

W does not know about the new conventions for detection of background jobs. It will sometimes find a background job instead of the right one.

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.

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 this message, preceded by 'Broadcast Message ...', to all logged in users.

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

FILES

/dev/tty?
/etc/utmp

SEE ALSO

mesg(1), write(1)

DIAGNOSTICS

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

NAME

wc − word count

SYNOPSIS

wc [**−lwcpt**] [**−b***baud*] [**−s***pagesize*] [**−u**] [**−v**] [name ...]

DESCRIPTION

Wc counts lines, words and characters, and optionally pages and the print time, 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 "lwcpt" is present, the specified counts (lines, words, characters, pages, or time) are selected by the letters **l**, **w**, **c**, **p**, or **t**. The default is **−lwc** unless **−v** is specified.

The **−b** option asks that the time be figured at the specified baud rate instead of the default 300 baud.

The **−s** option specifies that pages are *pagesize* lines long instead of the default 66.

The **−u** options asks that the time printed be based on uucp transmission time, about 90% as fast as normal.

The **−v** option asks for a verbose output format, with headers and including pages and time by default.

BUGS

The times given do not take into account variable factors such as system load; delays due to tab expansion or tty driver delays, which can be a factor with cu; or uucp delays such as mail headers, auxillary protocol files, or the time taken to initially connect to another site.

NAME

what — show what versions of object modules were used to construct a file

SYNOPSIS

what name ...

DESCRIPTION

What reads each file and searches for sequences of the form "@(#)" as inserted by the source code control system. It then prints the remainder of the string after this marker, up to a null character, newline, double quote, or ">" character.

BUGS

As SCCS is not licensed with UNIX/32V, this is a rewrite of the *what* command which is part of SCCS, and may not behave exactly the same as that command does.

NAME

whatis − describe what a command is

SYNOPSIS

whatis command ...

DESCRIPTION

Whatis looks up a given command and gives the header line from the manual section. You can then run the *man*(1) command to get more information. If the line starts 'name(section) ...' you can do 'man section name' to get the documentation for it. Try 'whatis ed' and then you should do 'man 1 ed' to get the manual.

Whatis is actually just the **−f** option to the *man*(1) command.

FILES

/usr/lib/whatis Data base

SEE ALSO

apropos(1), man(1), catman(8)

AUTHOR

William Joy

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.

FILES

/etc/utmp

SEE ALSO

getuid(2), utmp(5)

NAME

whoami − print effective current user id

SYNOPSIS

whoami

DESCRIPTION

Whoami prints who you are. It works even if you are su'd, while 'who am i' does not since it uses /etc/utmp.

FILES

/etc/passwd Name data base

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 is read from the terminal or an interrupt is sent. At that point *write* writes 'EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *ttyname* argument may be used to indicate the appropriate terminal name.

Permission to write may be denied or granted by use of the *mesg* command. At the outset writing is allowed. Certain commands, in particular *nroff* and *pr*(1) disallow messages in order to prevent messy output.

If the character '!' is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first write to another user, wait for him to write back before starting to send. Each party should end each message with a distinctive signal—**(o)** for 'over' is conventional—that the other may reply. **(oo)** for 'over and out' is suggested when conversation is about to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute '!'

SEE ALSO

mesg(1), who(1), mail(1)

NAME
xsend, xget, enroll — secret mail

SYNOPSIS
xsend person
xget
enroll

DESCRIPTION
These commands implement a secure communication channel; it is like *mail*(1), but no one can read the messages except the intended recipient. The method embodies a public-key cryptosystem using knapsacks.

To receive messages, use *enroll*; it asks you for a password that you must subsequently quote in order to receive secret mail.

To receive secret mail, use *xget*. It asks for your password, then gives you the messages.

To send secret mail, use *xsend* in the same manner as the ordinary mail command. (However, it will accept only one target). A message announcing the receipt of secret mail is also sent by ordinary mail.

FILES
/usr/spool/secretmail/*.key: keys /usr/spool/secretmail/*.[0-9]: messages

SEE ALSO
mail (1)

BUGS
It should be integrated with ordinary mail. The announcement of secret mail makes traffic analysis possible.

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. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

> **xstr −c** name

will extract the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file. The resulting C text is placed in the file *x.c,* to then be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffices of existing strings do not cause changes to the data base.

After all components of a large program have been compiled a file *xs.c* declaring the common *xstr* space can be created by a command of the form

> **xstr**

This *xs.c* file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. A command

> **xstr** name

creates files *x.c* and *xs.c* as before, without using or affecting any *strings* file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *Xstr* reads from its standard input when the argument '−' is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

> **cc −E** name.c | **xstr −c −**
> **cc −c** x.c
> **mv** x.o name.o

Xstr does not touch the file *strings* unless new items are added, thus *make* can avoid remaking *xs.o* unless truly necessary.

FILES

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array 'xstr'
/tmp/xs*	Temp file when 'xstr name' doesn't touch *strings*

SEE ALSO

mkstr(1)

AUTHOR

William 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
yes — be repetitively affirmative

SYNOPSIS
yes [**expletive**]

DESCRIPTION
Yes repeatedly outputs "y", or if **expletive** is given, that is output repeatedly. Termination is by rubout.

BUGS
Boring.

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.

(2J) System call entries added in support of the job control mechanisms of *csh*(1). These system calls are not available in standard Version 7 UNIX systems, and should be used only when necessary; to prevent inexplicit use they are contained in the *jobs* library which must be specifically requested with the **−ljobs** loader option. The use of conditional compilation is recommented when possible so that programs which use these features will gracefully degrade on systems which lack job control.

(2V) System calls added for the Virtual Memory version of UNIX distributed by Berkeley. Some of these calls are likely to be replaced by new facilities in future versions; in cases where this is imminent, this is indicated in the individual manual pages.

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.*

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 10240 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.

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 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 10^9 bytes).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek* was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 links to a file.

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

SEE ALSO

intro(3)

ASSEMBLER (PDP-11)

as /usr/include/sys.s file ...

The PDP11 assembly language interface is given for each system call. The assembler symbols are defined in '/usr/include/sys.s'.

Return values appear in registers r0 and r1; it is unwise to count on these registers being preserved when no value is expected. An erroneous call is always indicated by turning on the c-bit of the condition codes. The error number is returned in r0. The presence of an error is most easily tested by the instructions *bes* and *bec* ('branch on error set (or clear)'). These are synonyms for the *bcs* and *bcc* instructions.

On the Interdata 8/32, the system call arguments correspond well to the arguments of the C routines. The sequence is:

```
la      %2,errno
```

```
l       %0,&callno
svc     0,args
```

Thus register 2 points to a word into which the error number will be stored as needed; it is cleared if no error occurs. Register 0 contains the system call number; the nomenclature is identical to that on the PDP11. The argument of the *svc* is the address of the arguments, laid out in storage as in the C calling sequence. The return value is in register 2 (possibly 3 also, as in *pipe*) and is −1 in case of error. The overflow bit in the program status word is also set when errors occur.

On the VAX-11 a system call follows exactly the same conventions as a C procedure. Namely, register **ap** points to a long word containing the number of arguments, and the arguments follow in successive long words. Values are returned in registers **r0** and **r1**. An error is indicated by setting the C (carry) bit in the processor status word; the error number is placed in **r0**.

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;

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 (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 (PDP-11)

(access = 33.)
sys access; name; mode

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), sa(8)

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 (PDP-11)

(acct = 51.)
sys acct; file

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 2147483647 seconds.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), signal(2), sigsys(2), sigset(3), sleep(3)

ASSEMBLER (PDP-11)

(alarm = 27.)
(seconds in r0)
sys alarm
(previous amount in r0)

NAME

brk, sbrk, break — change core allocation

SYNOPSIS

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Brk sets the system's idea of the lowest location not used by the program (called the break) to *addr* (rounded up to the next multiple of 64 bytes on the PDP11, 256 bytes on the Interdata 8/32, and 1024 bytes on a VAX-11). Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function *sbrk, incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *break*.

The *vlimit*(2) system call may be used to determine the maximum permissible size of the *data* region; it will not be possible to set the break beyond "etext + vlimit(LIM_DATA, −1)." (See *end*(3) for the definition of *etext*.)

SEE ALSO

exec(2), vlimit(2), malloc(3), end(3)

DIAGNOSTICS

Zero is returned if the *brk* could be set; −1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break. *Sbrk* returns −1 if the break could not be set.

BUGS

Setting the break in the range 0177701 to 0177777 (on the PDP11) is the same as setting it to zero.

ASSEMBLER (PDP-11)

(break = 17.)
sys break; addr

Break performs the function of *brk*. The name of the routine differs from that in C for historical reasons.

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, the starting point for path names not beginning with '/'.

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.

ASSEMBLER

(chdir = 12.)
sys chdir; dirname

NAME

chmod − change mode of file

SYNOPSIS

chmod(name, mode)
char *name;

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode*. Modes are constructed by *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
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 (this is the default) 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. See *sticky*(8).

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

On some systems, writing or 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 (PDP-11)

(chmod = 15.)
sys chmod; name; mode

NAME

chown — change owner and group of a file

SYNOPSIS

chown(name, owner, group)
char *name;

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its *owner* and *group* changed as specified. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

On some systems, *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 (PDP-11)

(chown = 16.)
sys chown; name; owner; group

NAME

close — close a file

SYNOPSIS

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an *open, creat, dup* or *pipe*(2) call, *close* closes the associated file. A close of all files is automatic on *exit*, but since there is a limit on the number of open files per process, *close* is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain 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 (PDP-11)

(close = 6.)
(file descriptor in r0)
sys close

BUGS

A file cannot be closed while there are pages which have been *vread* but not referenced.

NAME

creat — create a new file

SYNOPSIS

creat(name, mode)
char *name;

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask (see *umask*(2)). Also see *chmod*(2) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat*, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write(2), close(2), chmod(2), umask (2)

DIAGNOSTICS

The value −1 is returned if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

ASSEMBLER (PDP-11)

(creat = 8.)
sys creat; name; mode
(file descriptor in r0)

BUGS

A file cannot be truncated while any process has pages set up by a *vread* on that file which have not been referenced.

NAME
dup, dup2 — duplicate an open file descriptor

SYNOPSIS
dup(fildes)
int fildes;

dup2(fildes, fildes2)
int fildes, fildes2;

DESCRIPTION
Given a file descriptor returned from an *open, pipe,* or *creat* call, *dup* allocates another file descriptor synonymous with the original. The new file descriptor is returned.

In the second form of the call, *fildes* is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). *Dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

SEE ALSO
creat(2), open(2), close(2), pipe(2)

DIAGNOSTICS
The value −1 is returned if: the given file descriptor is invalid; there are already too many open files.

ASSEMBLER (PDP-11)
(dup = 41.)
(file descriptor in r0)
(new file descriptor in r1)
sys dup
(file descriptor in r0)

The *dup2* entry is implemented by adding 0100 to *fildes*.

BUGS
Dup2 fails if *fildes2* was *vread* from and some of the pages have not been referenced.

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.

To aid execution of command files of various programs, if the first two characters of the executable file are '#!' then *exec* attempts to read a pathname from the executable file and use that program as the command files command interpreter. For example, the following command file sequence would be used to begin a *csh* script:

```
#! /bin/csh
# This shell script computes the checksum on /dev/foobar
#
        ...
```

A single parameter may be passed the interpreter, specified after the name of the interpreter; its length and the length of the name of the interpreter combined must not exceed 32 characters. The space (or tab) following the '#!' is mandatory, and the pathname must be explicit (no paths are searched).

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 (PDP-11)

(exec = 11.)
sys exec; name; argv

(exece = 59.)
sys exece; name; argv; envp

Plain *exec* is obsoleted by *exece*, but remains for historical reasons.

When the called file starts execution on the PDP11, 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.

```
sp→  nargs
     arg0
     ...
     argn
     0
     env0
     ...
```

```
        envm
        0
arg0:   <arg0\0>
        ...
envv0:  <env0\0>
        0
```

On the Interdata 8/32, the stack begins at a conventional place (currently 0xD0000) and grows upwards. After *exec*, the layout of data on the stack is as follows.

```
        int     0
arg0:   byte    ...
        ...
argp0:  int     arg0
        ...
        int     0
envp0:  int     env0
        ...
        int     0
%2→     space   40
        int     nargs
        int     argp0
        int     envp0
%3→
```

This arrangement happens to conform well to C calling conventions.

On a VAX-11, the stack begins at 0x7ffff000 and grows towards lower numbered addresses. After *exec*, the layout of data on the stack is as follows.

```
ap →
fp →
sp →   .long nargs
       .long arg0
       ...
       .long argn
       .long 0
       .long env0
       ...
       .long envn
       .long 0
arg0:  .byte "arg0\0"
       ...
envn:  .byte "envn\0"
       .long 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) or *vfork*(2) to avoid flushing buffered output twice.

SEE ALSO
fork(2), vfork(2), wait(2)

ASSEMBLER (PDP-11)
(exit = 1.)
(status in r0)
sys exit

NAME
fork — spawn new process

SYNOPSIS
fork()

DESCRIPTION
Fork and *vfork*(2) are the only ways new processes are 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 is used by *wait*(2).

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

Vfork is the most efficient way of creating a new process when the fork is to be followed shortly by an exec, but is not suitable when the fork is not to be followed by an exec.

SEE ALSO
wait(2), exec(2), vfork(2)

DIAGNOSTICS
Returns −1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.

ASSEMBLER (PDP-11)
(fork = 2.)
sys **fork**
(new process return)
(old process return, new process ID in r0)

The return locations in the old and new process differ by one 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 (PDP-11)

(getpid = 20.)
sys getpid
(pid in r0)

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 contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the 'set user ID' mode, to find out who invoked them.

Getgid returns the real group ID, *getegid* the effective group ID.

SEE ALSO

setuid(2)

ASSEMBLER (PDP-11)

(getuid = 24.)
sys getuid
(real user ID in r0, effective user ID in r1)

(getgid = 47.)
sys getgid
(real group ID in r0, effective group ID in r1)

NAME

ioctl, stty, gtty − control device

SYNOPSIS

#include <sgtty.h>

ioctl(fildes, request, argp)
struct sgttyb *argp;

stty(fildes, argp)
struct sgttyb *argp;

gtty(fildes, argp)
struct sgttyb *argp;

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). 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 is peculiar to the Berkeley implementation, and 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.

Ioctl calls which attempt to modify the state of a process control terminal while a process is not in the process group of the control terminal will cause a SIGTTOU signal to be sent to the process' process group. Such *ioctls* are allowed, however, if SIGTTOU is being held, ignored, if the process is an orphan which has been inherited by *init*, or is the child in an incomplete *vfork* (see *jobs*(3))

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 (PDP-11)

(ioctl = 54.)
sys ioctl; fildes; request; argp

(stty = 31.)
(file descriptor in r0)
stty; argp

(gtty = 32.)
(file descriptor in r0)
sys gtty; argp

NAME

kill − send signal to a process

SYNOPSIS

kill(pid, sig)

DESCRIPTION

Kill sends the signal *sig* to the process specified by the process number *pid.* See *sigsys*(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. (A single exception is the signal SIGCONT which may be sent as described in *killpg*(2), although it is usually sent using *killpg* rather than *kill*).

If the process number is 0, the signal is sent to all other processes in the sender's process group; see *tty*(4) and also *killpg*(2).

If the process number is −1, and the user is the super-user, the signal is broadcast universally except to processes 0, 1, 2, the scheduler initialization, and pageout processes, and the process sending the signal.

Processes may send signals to themselves.

SEE ALSO

sigsys(2), signal(2), kill(1), killpg(2), init(8)

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 (PDP-11)

(kill = 37.)
(process number in r0)
sys kill; sig

NAME

killpg — send signal to a process or a process group

SYNOPSIS

killpg(pgrp, sig)

cc ... −ljobs

DESCRIPTION

Killpg sends the signal *sig* to the specified process group. See *sigsys*(2) for a list of signals; see *jobs*(3) for an explanation of process groups.

The sending process and members of the process group must have the same effective user ID, otherwise this call is restricted to the super-user. As a single special case the continue signal SIGCONT may be sent to any process which is a descendant of the current process. This allows a command interpreter such as *csh*(1) to restart set-user-id processes stopped from the keyboard with a stop signal.

The calls

killpg(0, sig)

and

kill(0, sig)

have identical effects, sending the signal to all members of the invoker's process group (including the process itself). It is preferable to use the call involving *kill* in this case, as it is portable to other UNIX systems.

SEE ALSO

jobs(3), kill(2), sigsys(2), signal(2), csh(1), kill(1)

DIAGNOSTICS

Zero is returned if the processes are sent the signals; −1 is returned if any process in the process group cannot be sent the signal, or if there are no members in the process group.

BUGS

The job control facilities are not available in standard version 7 UNIX. These facilities are still under development and may change in future releases of the system as better inter-process communication facilities and support for virtual terminals become available. The options and specifications of this system call and even the call itself are thus subject to change.

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.

SEE ALSO

ln(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; −1 is returned when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when a file has too many links.

On some systems the super-user may link to non-ordinary files.

ASSEMBLER (PDP-11)

(link = 9.)
sys link; name1; name2

NAME

lseek, tell — move read/write pointer

SYNOPSIS

long lseek(fildes, offset, whence)
long offset;

long tell(fildes)

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (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 obsolete function *tell(fildes)* is identical to *lseek(fildes, 0L, 1)*.

Seeking far beyond the end of a file, then writing, creates a gap or 'hole', which occupies no physical space and reads as zeros.

SEE ALSO

open(2), creat(2), fseek(3)

DIAGNOSTICS

−1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

BUGS

Lseek is a no-op on character special files.

ASSEMBLER (PDP-11)

(lseek = 19.)
(file descriptor in r0)
sys lseek; offset1; offset2; whence

Offset1 and *offset2* are the high and low words of *offset*; r0 and r1 contain the pointer upon return.

NAME

mknod — make a directory or a special file

SYNOPSIS

mknod(name, mode, addr)
char *name;

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)). The first block pointer of the i-node is initialized from *addr*. For ordinary files and directories *addr* is normally zero. In the case of a special file, *addr* specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(1), mknod(1), filsys(5)

DIAGNOSTICS

Zero is returned if the file has been made; −1 if the file already exists or if the user is not the super-user.

ASSEMBLER (PDP-11)

(mknod = 14.)
sys mknod; name; mode; addr

NAME

mount, umount — mount or remove file system

SYNOPSIS

mount(special, name, rwflag)
char *special, *name;

umount(special)
char *special;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special;* from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. *Name* must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

SEE ALSO

mount(8)

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 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 (PDP-11)

(mount = 21.)
sys mount; special; name; rwflag

(umount = 22.)
sys umount; special

NAME

mpx — create and manipulate multiplexed files

SYNOPSIS

mpx(name, access)
char *name;

join(fd, xd)

chan(xd)

extract(i, xd)

attach(i, xd)

detach(i, xd)

connect(fd, cd, end)

npgrp(i, xd, pgrp)

ckill(i, xd, signal)

#include <sys/mx.h>
mpxcall(cmd, vec)
int *vec;

DESCRIPTION

mpxcall(cmd, vec) is the system call shared by the library routines described below. *Cmd* selects a command using values defined in *<sys/mx.h>*. *Vec* is the address of a structure containing the arguments for the command.

mpx(name, access)

Mpx creates and opens the file *name* with access permission *access* (see *creat*(2)) and returns a file descriptor available for reading and writing. A −1 is returned if the file cannot be created, if *name* already exists, or if the file table or other operating system data structures are full. The file descriptor is required for use with other routines.

If *name* is 0, a file descriptor is returned as described but no entry is created in the file system.

Once created an mpx file may be opened (see *open*(2)) by any process. This provides a form of interprocess communication whereby a process B can 'call' process A by opening an mpx file created by A. To B, the file is ordinary with one exception: the *connect* primitive could be applied to it. Otherwise the functions described below are used only in process A and descendants that inherit the open mpx file.

When a process opens an mpx file, the owner of the file receives a control message when the file is next read. The method for 'answering' this kind of call involves using *attach* and *detach* as described in more detail below.

Once B has opened A's mpx file it is said to have a *channel* to A. A channel is a pair of data streams: in this case, one from B to A and the other from A to B. Several processes may open the same mpx file yielding multiple channels within the one mpx file. By accessing the appropriate channel, A can communicate with B and any others. When A reads (see *read*(2)) from the mpx file data written to A by the other processes appears in A's buffer using a record format described in *mpxio*(5). When A writes (see *write*(2)) on its mpx file the data must be formatted in a similar way.

The following commands are used to manipulate mpx files and channels.

> *join*— adds a new channel on an mpx file to an open file F. I/O on the new channel is I/O on F.
> *chan*— creates a new channel.

extract— file descriptor maintenance.
connect— similar to join except that the open file F is connected to an existing channel.
attach and *detach*— used with call protocol.
npgrp— manipulates process group numbers so that a channel can act as a control terminal (see *tty*(4)).
ckill— send signal (see *signal*(2)) to process group through channel.

A maximum of 15 channels may be connected to an mpx file. They are numbered 0 through 14. *Join* may be used to make one mpx file appear as a channel on another mpx file. A hierarchy or tree of mpx files may be set up in this way. In this case one of the mpx files must be the root of a tree where the other mpx files are interior nodes. The maximum depth of such a tree is 4.

An *index* is a 16-bit value that denotes a location in an mpx tree other than the root: the path through mpx 'nodes' from the root to the location is expressed as a sequence of 4-bit nibbles. The branch taken at the root is represented by the low-order 4-bits of an index. Each succeeding branch is specified by the next higher-order nibble. If the length of a path to be expressed is less than 4, then the illegal channel number, 15, must be used to terminate the sequence. This is not strictly necessary for the simple case of a tree consisting of only a root node: its channels can be expressed by the numbers 0 through 14. An index *i* and file descriptor *xd* for the root of an mpx tree are required as arguments to most of the commands described below. Indices also serve as channel identifiers in the record formats given in *mpxio*(5). Since −1 is not a valid index, it can be returned as a error indication by subroutines that normally return indices.

The operating system informs the process managing an mpx file of changes in the status of channels attached to the file by generating messages that are read along with data from the channels. The form and content of these messages is described in *mpxio*(5).

join(fd, xd) establishes a connection (channel) between an mpx file and another object. *Fd* is an open file descriptor for a character device or an mpx file and *xd* is the file descriptor of an mpx file. *Join* returns the index for the new channel if the operation succeeds and −1 if it does not.

Following join, *fd* may still be used in any system call that would have been meaningful before the join operation. Thus a process can read and write directly to *fd* as well as access it via *xd*. If the number of channels required for a tree of mpx files exceeds the number of open files permitted a process by the operating system, some of the file descriptors can be released using the standard *close*(2) call. Following a close on an active file descriptor for a channel or internal mpx node, that object may still be accessed through the root of the tree.

chan(xd) allocates a channel and connects one end of it to the mpx file represented by file descriptor *xd*. *Chan* returns the index of the new channel or a −1 indicating failure. The *extract* primitive can be used to get a non-multiplexed file descriptor for the free end of a channel created by *chan*.

Both *chan* and *join* operate on the mpx file specified by *xd*. File descriptors for interior nodes of an mpx tree must be preserved or reconstructed with *extract* for use with *join* or *chan*. For the remaining commands described here, *xd* denotes the file descriptor for the root of an mpx tree.

extract(i, xd) returns a file descriptor for the object with index *i* on the mpx tree with root file descriptor *xd*. A −1 is returned by extract if a file descriptor is not available or if the arguments do not refer to an existing channel and mpx file.

attach(i, xd)
detach(i, xd). If a process A has created an mpx file represented by file descriptor *xd*, then a process B can open (see *open*(2)) the mpx file. The purpose is to establish a channel between

A and B through the mpx file. *Attach* and *Detach* are used by A to respond to such opens.

An open request by B fails immediately if a new channel cannot be allocated on the mpx file, if the mpx file does not exist, or if it does exist but there is no process (A) with a multiplexed file descriptor for the mpx file (i.e. *xd* as returned by *mpx*(2)). Otherwise a channel with index number *i* is allocated. The next time A reads on file descriptor *xd*, the WATCH control message (see *mpxio*(5)) will be delivered on channel *i*. A responds to this message with *attach* or *detach*. The former causes the open to complete and return a file descriptor to B. The latter deallocates channel *i* and causes the open to fail.

One mpx file may be placed in 'listener' mode. This is done by writing *ioctl(xd, MXLSTN, 0)* where *xd* is an mpx file descriptor and MXLSTN is defined in */usr/include/sgtty.h*. The semantics of listener mode are that all file names discovered by *open*(2) to have the syntax *system!pathname* (see *uucp*(1)) are treated as opens on the mpx file. The operating system sends the listener process an OPEN message (see *mpxio*(5)) which includes the file name being opened. *Attach* and *detach* then apply as described above.

Detach has two other uses: it closes and releases the resources of any active channel it is applied to, and should be used to respond to a CLOSE message (see *mpxio*(5)) on a channel so the channel may be reused.

connect(fd, cd, end). *Fd* is a character file descriptor and *cd* is a file descriptor for a channel, such as might be obtained via *extract(chan(xd), xd)* or by *open*(2) followed by *attach*. *Connect* splices the two streams together. If *end* is negative, only the output of *fd* is spliced to the input of *cd*. If *end* is positive, the output of *cd* is spliced to the input of *fd*. If *end* is zero, then both splices are made.

npgrp(i, xd, pgrp). If *xd* is negative *npgrp* applies to the process executing it, otherwise *i* and *xd* are interpreted as a channel index and mpx file descriptor and *npgrp* is applied to the process on the non-multiplexed end of the channel. If *pgrp* is zero, the process group number of the indicated process is set to the process number of that process, otherwise the value of *pgrp* is used as the process group number.

Npgrp normally returns the new process group number. If *i* and *xd* specify a nonexistent channel, *npgrp* returns −1.

ckill(i, xd, signal) sends the specified signal (see *signal*(2)) through the channel specified by *i* and *xd*. If the channel is connected to anything other than a process, *ckill* is a null operation. If there is a process at the other end of the channel, the process group will be interrupted (see *signal*(2), *kill*(2)). *Ckill* normally returns *signal*. If *ch* and *xd* specify a nonexistent channel, *ckill* returns −1.

FILES

/usr/include/sys/mx.h
/usr/include/sgtty.h

SEE ALSO

mpxio(5)

BUGS

Mpx files are an experimental part of the operating system more subject to change and prone to bugs than other parts.

Maintenance programs, e.g. *icheck*(1), diagnose mpx files as an illegal mode.

Channels may only be connected to objects in the operating system that are accessible through the line discipline mechanism.

Higher performance line disciplines are needed.

The maximum tree depth restriction is not really checked.

A non-destructive *disconnect* primitive (inverse of *connect*) is not provided.

A non-blocking flow control strategy based on messages defined in *mpxio*(5) should not be attempted by novices; the enabling *ioctl* command should be protected.

The *join* operation could be subsumed by *connect*. A mechanism is needed for moving a channel from one location in an mpx tree to another.

NAME

nice − set program priority

SYNOPSIS

nice(incr)

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range −20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by *fork*(2). For a privileged process to return to normal priority from an unknown state, *nice* should be called successively with arguments −40 (goes to priority −20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO

nice(1), fork(2), renice(8)

ASSEMBLER (PDP-11)

(nice = 34.)
(priority in r0)
sys **nice**

NAME

open − open for reading or writing

SYNOPSIS

open(name, mode)
char *name;

DESCRIPTION

Open opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The 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. writable), or if too many files are open.

ASSEMBLER (PDP-11)

(open = 5.)
sys open; name; mode
(file descriptor in r0)

BUGS

It should be possible to optionally open files for writing with exclusive use, and to optionally call *open* without the possibility of hanging waiting for carrier on communication lines.

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), sigsys(2), signal(2), sigset(3), setjmp(3)

ASSEMBLER (PDP-11)

(pause = 29.)
sys pause

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 (PDP-11)

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

NAME

profil — execution time profile

SYNOPSIS

profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777(8) gives a 1-1 mapping of pc's to words in *buff*; 077777(8) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling may be turned off if an update in *buff* would cause a memory fault.

SEE ALSO

monitor(3), prof(1)

ASSEMBLER (PDP-11)

(profil = 44.)
sys profil; buff; bufsiz; offset; scale

BUGS

Profiling does not work for interpreters; if a signal were given to a process when its cpu-time clock ticked then profiling interpreters would be possible.

NAME

ptrace – process trace

SYNOPSIS

#include <signal.h>

ptrace(request, pid, addr, data)
int *addr;

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt.' See *signal*(2) for the list. Then the traced process enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated, request 1 indicates I space, 2 D space. *Addr* must be even. The child must be stopped. The input *data* is ignored.

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

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. (On the PDP-11 and VAX-11 the T-bit is used and just one instruction is executed; on the Interdata the stop does not take place until a store instruction is executed.) This is part of the mechanism for implementing breakpoints.

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.

On the Interdata 8/32, 'word' means a 32-bit word and 'even' means 0 mod 4. On a VAX-11, 'word' also means a 32-bit integer, but the 'even' restriction does not apply.

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.

On the Interdata 8/32, 'as soon as possible' (request 7) means 'as soon as a store instruction has been executed.'

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

(ptrace = 26.)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)

NAME

read — read from file

SYNOPSIS

read(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached.

Unless the reader is ignoring or holding SIGTTIN signals, reads from the control typewriter while not in its process group cause a SIGTTIN signal to be sent to the reader's process group; in the former case an end-of-file is returned.

SEE ALSO

open(2), creat(2), dup(2), pipe(2), vread(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 (PDP-11)

(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes
(byte count in r0)

BUGS

It should be possible to call *read* and have it return immediately without blocking if there is no input available. As a single special case, this is currently done on control terminals when the reading process has requested SIGTINT signals when input arrives (see *tty*(4)).

Processes which have been orphaned by their parents and have been inherited by *init*(8) never receive SIGTTIN signals. Instead *read* returns with an end-of-file indication.

NAME

reboot — reboot system or halt processor

SYNOPSIS

#include <sys/reboot.h>

reboot(howto)
int howto;

DESCRIPTION

Reboot is used to cause a system reboot, and is invoked automatically in the event of unrecoverable system failures. *Howto* is a mask of options passed to the bootstrap program. The system call interface permits only RB_HALT or RB_AUTOBOOT to be passed to the reboot program; the other flags are used in scripts stored on the console storage media, or used in manual bootstrap procedures. When none of these options (e.g. RB_AUTOBOOT) is given, the system is rebooted from file "vmunix" in the root file system of unit 0 of a disk chosen in a processor specific way: on the 11/780 it is specified by a line in the DEFBOO.CMD script on the console floppy; on the 11/750 it is determined by the setting of the front panel switch which picks the bootstrap device. An automatic consistency check of the disks is then normally performed.

The bits of *howto* are:

RB_HALT
the processor is simply halted; no reboot takes place. This should be used with caution.

RB_ASKNAME
Interpreted by the bootstrap program itself, causing it to inquire as to what file should be booted. Normally, the system is booted from the file "xx(0,0)vmunix" without asking, where *xx* is determined by a code in register *r10* (which is known as *devtype)* at entry to the bootstrap program. The code corresponds to the major device number of the root file system, i.e. "major(rootdev)". Currently, the following values of *devtype* are understood:

0	hp	rm03/rm05/rm80/rp06 massbus disk
1	--	unused
2	up	unibus disks (emulex sc21 w/ cdc/ampex/fujitsu drives)
3	rk	rk07 unibus disks

Thus if *r10* contained a 2, the system

up(0,0)vmunix.

would be booted. This switch not available from the system call interface.

RB_SINGLE
Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. This prevents the consistency check, rather simply booting the system with a single-user shell on the console, from the file system specified by *r10*. This switch is interpreted by the *init*(8) program in the newly booted system. This switch is not available from the system call interface.

SEE ALSO

crash(8), halt(8), init(8), reboot(8)

BUGS

NAME

setpgrp, getpgrp — set/get process group

SYNOPSIS

int getpgrp(pid)

setpgrp(pid, pgrp)

cc ... −ljobs

DESCRIPTION

The process group of the specified process is returned by *getpgrp. Setpgrp* sets the process group of the specified process *pid* to the specified *pgrp.* If *pid* is zero, then the call applies to the current process.

If the invoker is not the super-user, then the affected process must have the same effective user-id as the invoker or be a descendant of the invoking process.

This call is used by *csh*(1) to create process groups in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in *tty*(4) are used to get/set the process group of the control terminal.

See *jobs(3)* for a general discussion of job control.

SEE ALSO

jobs(3), getuid(2), tty(4)

BUGS

The job control facilities are not available in standard version 7 UNIX. These facilities are still under development and may change in future releases of the system as better inter-process communication facilities and support for virtual terminals become available. The options and specifications of these system calls and even the calls themselves are thus subject to change.

A system call *setpgrp* has been implemented in other versions of UNIX which are not widely used outside of Bell Laboratories; these implementations have, in general, slightly different semantics.

NAME
setuid, setgid − set user and group ID

SYNOPSIS
setuid(uid)

setgid(gid)

DESCRIPTION
The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user 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 (PDP-11)
(setuid = 23.)
(user ID in r0)
sys setuid

(setgid = 46.)
(group ID in r0)
sys setgid

NAME

signal — catch or ignore signals

SYNOPSIS

#include <signal.h>

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

DESCRIPTION

N.B.: The system currently supports two signal implementations. The one described here is standard in version 7 UNIX systems, and is retained for backward compatabililty. The one described in *sigsys*(2) as supplemented by *sigset*(3) provides for the needs of the job control mechanisms used by *csh*(1), and corrects the bugs in this older implementation of signals, allowing programs which process interrupts to be written reliably.

A signal is generated by some abnormal event, initiated either by user at a terminal (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 with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

N.B.: There are actually more signals; see *sigsys*(2); the signals listed here are those of standard version 7.

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.

If, when using this (older) signal interface, a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during an *ioctl*, *read*, or *write*(2) on a slow device (like a terminal; 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.

If a process is using the mechanisms of *sigsys*(2) and *sigset*(3) then many of these calls are automatically restarted (See *sigsys*(2) and *jobs*(3) for details).

SEE ALSO

sigsys(2), kill(1), kill(2), ptrace(2), setjmp(3), sigset(3)

DIAGNOSTICS

The value (int)−1 is returned if the given signal is out of range.

BUGS

The traps should be distinguishable by extra arguments to the signal handler, and all hardware supplied parameters should be made available to the signal routine.

If a repeated signal arrives before the last one can be reset, there is no chance to catch it (however this is **not** true if you use *sigsys*(2) and *sigset*(3)).

The type specification of the routine and its *func* argument are problematical.

ASSEMBLER (PDP-11)

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

If *label* is 0, default action is reinstated. If *label* is 1, the signal is ignored. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt.

NOTES (VAX-11)

See *sigsys*(2) for information on how hardware faults are mapped into signals.

NAME

sigsys — catch or ignore signals

SYNOPSIS

#include <signal.h>

(*sigsys(sig, func))()
void (*func)();

cc ... −ljobs

DESCRIPTION

N.B.: The system currently supports two signal implementations. The one described in *signal*(2) is standard in version 7 UNIX systems, and retained for backward compatibility as it is different in a number of ways. The one described here (with the interface in *sigset*(3)) provides for the needs of the job control mechanisms (see *jobs*(3)) used by *csh*(1), and corrects the bugs in the standard implementation of signals, allowing programs which process interrupts to be written reliably.

The routine *sigsys* is not normally called directly; rather the routines of *sigset*(3) should be used. These routines are kept in the "jobs" library, accessible by giving the loader option **−ljobs**. The features described here are less portable then those of *signal*(2) and should not be used in programs which are to be moved to other versions of UNIX.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty*(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals which cannot be blocked, the *sigsys* call allows signals either to be ignored, held until a later time (protecting critical sections in the process), or to cause an interrupt to a specified location. Here is the list of all 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, held 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
SIGSTOP	17†	stop (cannot be caught, held or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal

SIGTTOU	22†	background write attempted to control terminal
SIGTINT	23•	input record is available at control terminal
SIGXCPU	24	cpu time limit exceeded (see *vlimit*(2))
SIGXFSZ	25	file size limit exceeded (see *vlimit*(2))

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

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If *func* is SIG_HOLD the signal is remembered if it occurs, but not presented to the process; it may be presented later if the process changes the action for the signal. If *func* is SIG_IGN the signal is subsequently ignored, and pending instances of the signal are discarded (i.e. if the action was previously SIG_HOLD.) Otherwise when the signal occurs *func* will be called.

A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal, set with *sigsys*, is reset to SIG_DFL after being caught. However by specifying DEFERSIG(func) as the last argument to *sigsys*, one causes the action to be set to SIG_HOLD before the interrupt is taken, so that recursive instances of the signal cannot occur during handling of the signal.

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 terminal; but not a file) and during a *pause* or *wait*(2). When a signal occurs during one of these calls, 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. *Read* and *write* calls which have done no I/O, *ioctls* blocked with SIGTTOU, and *wait3* calls are restarted.

The value of *sigsys* is the previous (or initial) value of *func* for the particular signal.

The system provides two other functions by oring bits into the signal number: SIGDOPAUSE causes the process to *pause* after changing the signal action. It can be used to atomically re-enable a held signal which was being processed and wait for another instance of the signal. SIGDORTI causes the system to simulate an *rei* instruction clearing the mark the system placed on the stack at the point of interrupt before checking for further signals to be presented due to the specified change in signal actions. This allows a signal package such as *sigset*(3) to dismiss from interrupts cleanly removing the old state from the stack before another instance of the interrupt is presented.

After a *fork*(2) or *vfork*(2) the child inherits all signals. *Exec*(2) resets all caught signals to default action; held signals remain held and ignored signals remain ignored.

SEE ALSO

kill(1), ptrace(2), kill(2), jobs(3), sigset(3), setjmp(3), tty(4)

DIAGNOSTICS

The value BADSIG is returned if the given signal is out of range.

BUGS

The job control facilities are not available in standard version 7 UNIX. These facilities are still under development and may change in future releases of the system as better inter-process communication facilities and support for virtual terminals become available. The options and specifications of this facility and the system calls supporting it are thus subject to change.

Since only one signal action can be changed at a time, it is not possible to get the effect of SIGDOPAUSE for more than one signal at a time.

The traps (listed below) should be distinguishable by extra arguments to the signal handler, and all hardware supplied parameters should be made available to the signal routine.

ASSEMBLER (PDP-11)

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

If *label* is 0, default action is reinstated. If *label* is 1, the signal is ignored. If *label* is 3, the signal is held. Any other even *label* specifies an address in the process where an interrupt is simulated. If label is otherwise odd, the signal is sent to the function whose address is the label with the low bit cleared with the action set to SIG_HOLD. (Thus DEFERSIG is indicated by the low bit of a signal catch address. An RTI or RTT instruction will return from the interrupt.)

NOTES (VAX-11)

The handler routine can be declared:

handler(signo, param, xx, pc, psl)

Here *signo* is the signal name, into which the hardware faults and traps are mapped as defined below. Param is the parameter which is either a constant as given below or, for compatibility mode faults, the code provided by the hardware. Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl.

The routine is actually called with only 3 parameters specified in the *calls* or *callg* instruction. After return from the signal handler the *pc* and *psl* are popped off of the stack with an *rei*, so they act as "value-result" parameters unlike normal C value parameters.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <signal.h>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	
Protection violation	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_PRIVIN_FAULT
Reserved addressing	SIGILL	ILL_RESOP_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	
Chmu	SIGSEGV	

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)
struct stat *buf;
```

DESCRIPTION

Stat obtains detailed information about a named file. *Fstat* obtains the same information about an open file known by the file descriptor from a successful *open, creat, dup* or *pipe*(2) call.

Name points to a null-terminated string naming a file; *buf* is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable. The layout of the structure pointed to by buf as defined in <*stat.h*> is given below. *St_mode* is encoded according to the '#define' statements.

```
/*	stat.h	4.2	81/02/19	*/

struct	stat
{
	dev_t	st_dev;
	ino_t	st_ino;
	unsigned short st_mode;
	short	st_nlink;
	short	st_uid;
	short	st_gid;
	dev_t	st_rdev;
	off_t	st_size;
	time_t	st_atime;
	time_t	st_mtime;
	time_t	st_ctime;
};

#define	S_IFMT	0170000		/* type of file */
#define		S_IFDIR	0040000	/* directory */
#define		S_IFCHR	0020000	/* character special */
#define		S_IFBLK	0060000	/* block special */
#define		S_IFREG	0100000	/* regular */
#define		S_IFMPC	0030000	/* multiplexed char special */
#define		S_IFMPB	0070000	/* multiplexed block special */
#define	S_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 an i-node number, restricted permissions, and a not necessarily meaningful length.

st_atime is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. *st_mtime* is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. *st_ctime* is set both both by writing and changing the i-node.

SEE ALSO

ls(1), filsys(5)

DIAGNOSTICS

Zero is returned if a status is available; −1 if the file cannot be found.

ASSEMBLER

(stat = 18.)
sys stat; name; buf

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf

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 (PDP-11)

(stime = 25.)
(time in r0-r1)
sys stime

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,* etc. It is mandatory before a boot.

SEE ALSO

sync(1), update(8)

BUGS

The writing, although scheduled, is not necessarily complete upon return from *sync*.

ASSEMBLER (PDP-11)

(sync = 36.)
sys sync

NAME

syscall – indirect system call

SYNOPSIS

syscall(number, r0, r1, arg ...) (PDP-11)
syscall(number, arg, ...) (VAX-11)

DESCRIPTION

Syscall performs the system call whose assembly language interface has the specified *number*, register arguments *r0* and *r1* (on the PDP-11, regardless of whether the entry point really uses them) and further arguments *arg*.

The r0 value of the system call is returned.

DIAGNOSTICS

When the C-bit is set, *syscall* returns –1 and sets the external variable *errno* (see *intro*(2)).

BUGS

There is no way to simulate system calls such as *pipe*(2), which return values in register r1.

ASSEMBLER (PDP-11)

(indir = 0.)
sys indir; call

The system call at the location *call* is executed. Execution resumes after the *indir* call.

On the PDP-11, the main purpose of *indir* is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If *indir* is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, *indir* returns error code EINVAL; see *intro*(2).

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>*:

```
/*      timeb.h   3.26/6/80*/

/*
 * Structure returned by ftime system call
 */
struct timeb
{
        time_t    time;
        unsigned short millitm;
        short     timezone;
        short     dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local 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
date(1), stime(2), ctime(3)

ASSEMBLER (PDP-11)
(ftime = 35.)
sys ftime; bufptr

(time = 13.; obsolete call)
sys time
(time since 1970 in r0-r1)

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/HZ seconds, where HZ is either 50 or 60 depending on your locality.

This is the structure returned by *times*:

```
/*	times.h 4.1	11/9/80	*/

/*
 * Structure returned by times()
 */
struct tms {
	time_t	tms_utime;		/* user time */
	time_t	tms_stime;		/* system time */
	time_t	tms_cutime;		/* user time, children */
	time_t	tms_cstime;		/* system time, children */
};
```

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

SEE ALSO

time(1), time(2), vtimes(2)

ASSEMBLER (PDP-11)

(times = 43.)
sys times; buffer

NAME

umask — set file creation mode mask

SYNOPSIS

umask(complmode)

DESCRIPTION

Umask sets a mask used whenever a file is created by *creat*(2) or *mknod*(2): the actual mode (see *chmod*(2)) of the newly-created file is the logical **and** of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 022 (write access for owner only). The mask is inherited by child processes.

SEE ALSO

creat(2), mknod(2), chmod(2)

ASSEMBLER (PDP-11)

(umask = 60.)
sys umask; complmode

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.

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 (PDP-11)

(unlink = 10.)
sys unlink; name

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 *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 (PDP-11)

(utime = 30.)
sys utime; file; timep

NAME

vadvise − give advice to paging system

SYNOPSIS

vadvise(param)

DESCRIPTION

Vadvise is used to inform the system that process paging behavior merits special consideration. Parameters to *vadvise* are defined in the file **<vadvise.h>** . Currently, two calls to *vadvise* are implemented:

The call

vadvise(VA_ANOM);

advises that the paging behavior is not likely to be well handled by the system's default algorithm, since reference information collected over macroscopic intervals (e.g. 10-20 seconds) will not serve to indicate future page references. The system in this case will choose to replace pages with little emphasis placed on recent usage, and more emphasis on referenceless circular behavior. It is *essential* that processes which have very random paging behavior (such as LISP during garbage collection of very large address spaces) call *vadvise,* as otherwise the system has great difficulty dealing with their page-consumptive demands.

The call

vadvise(VA_NORM);

restores default paging replacement behavior after a call to

vadvise(VA_ANOM);

BUGS

This call is peculiar to this version of UNIX. The options and specifications of this system call and even the call itself are expected to change. It is expected to be extended with additional facilities in future versions of the system. In particular it is expected that this call will be particular to a segment, and that other behaviors such as sequential behavior will be specifiable.

NAME

vfork — spawn new process in a virtual memory efficient way

SYNOPSIS

vfork()

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process, which is horrendously inefficient in a paged environment. It is useful when the purpose of *fork*(2) would have been to create a new system context for an *exec*. *Vfork* differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *exec*(2) or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like *fork*. It does not work, however, to return while running in the childs context from the procedure which called *vfork* since the eventual return from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit* if you can't *exec*, since *exit* will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

Similarly when using the new signal mechanism of *sigset*(3) mechanism be sure to call *sigsys* rather than *signal*(2).

SEE ALSO

fork(2), exec(2), sigsys(2), wait(2),

DIAGNOSTICS

Same as for *fork*.

BUGS

This system call may be unnecessary if the system sharing mechanisms allow *fork* to be implemented more efficiently; users should not depend on the memory sharing semantics of *vfork* as it could, in that case, be made synonymous to *fork*.

To avoid a possible deadlock situation, processes which are children in the middle of a *vfork* are never sent SIGTTOU or SIGTTIN signals; rather, output or *ioctls* are allowed and input attempts result in an end-of-file indication.

This call is peculiar to this version of UNIX.

NAME

vhangup — virtually "hangup" the current control terminal

SYNOPSIS

vhangup()

DESCRIPTION

Vhangup is used by the initialization process *init*(8) to arrange that users are given "clean"' terminals at login, by revoking access of the previous users' processes to the terminal. To effect this, *vhangup* searches the system tables for references to the control terminal of the invoking process, revoking access permissions on each instance of the terminal which it finds. Further attempts to access the terminal by the affected processes will yield i/o errors (EBADF). Finally, a hangup signal (SIGHUP) is sent to the process group of the control terminal.

SEE ALSO

init (8)

BUGS

Access to the control terminal via **/dev/tty** is still possible.

This call is peculiar to this version of UNIX. The options and specifications of this system call and even the call itself are subject to change.

NAME

vlimit − control maximum system resource consumption

SYNOPSIS

#include <sys/vlimit.h>

vlimit(resource, value)

DESCRIPTION

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified *resource.* If *value* is specified as −1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE
: A pseudo-limit; if set non-zero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.

LIM_CPU
: the maximum number of cpu-seconds to be used by each process

LIM_FSIZE
: the largest single file which can be created

LIM_DATA
: the maximum growth of the data+stack region via *sbrk*(2) beyond the end of the program text

LIM_STACK
: the maximum size of the automatically-extended stack region

LIM_CORE
: the size of the largest core dump that will be created.

LIM_MAXRSS
: a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared LIM_MAXRSS.

Because this information is stored in the per-process information this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to *csh*(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way; a *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal!).

A file i/o operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the cpu time limit.

SEE ALSO

csh(1)

BUGS

If LIM_NORAISE is set, then no grace should be given when the cpu time limit is exceeded.

There should be *limit* and *unlimit* commands in *sh*(1) as well as in *csh.*

This call is peculiar to this version of UNIX. The options and specifications of this system call and even the call itself are subject to change. It may be extended or replaced by other facilities in future versions of the system.

NAME

vread — read virtually

SYNOPSIS

vread (fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

N.B.: This call is likely to be replaced by more general virtual memory facilities in the near future.

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* will be read (see *read*(2)). In particular, if the returned value is 0, then end-of-file has been reached.

Unlike *read*(2), *vread* does not necessarily or immediately fetch the data requested from *fildes*, but merely insures that the data will be fetched from the file descriptor *sometime before* the first reference to the data, at the system's discretion. Thus *vread* allows the system, among other possibilities, to choose to read data on demand, with whatever granularity is allowed by the memory management hardware, or to just read it in immediately as with *read*. A companion *vwrite*(2) call may be used with *vread* to provide an efficient mechanism for updating large files. The behavior of *vread* if other processes are writing to *fildes* is not defined.

Both the address of *buffer* and the current offset in *fildes* (as told by *tell*(2)) must be aligned to a multiple of VALSIZ (defined in **<valign.h>**). The library routine *valloc*(3) allocates properly aligned blocks from the free list.

Note for non-virtual systems: the *vread* system call can be simulated (exactly, if less efficiently) by *read*. If the unit on which a *vread* is done is not capable of supporting efficient demand initialization (e.g. a terminal or a pipe), then the system may choose to treat a call to *vread* as if it were a call to *read* at its discretion.

SEE ALSO

read(2), write(2), vwrite (2), valloc(3)

DIAGNOSTICS

A 0 is returned at end-of-file. If the read was otherwise unsuccessful, a -1 is returned. Physical I/O errors, non-aligned or bad buffer addresses, preposterous *nbytes*, file descriptor not that of an input file, and file offset not properly aligned can all generate errors.

BUGS

You can't *close* a file descriptor which you have *vread* from while there are still pages in the file which haven't been fetched by the system into your address space. In no case can a file descriptor which had such pages at the point of a *vfork* be closed during the *vfork*.

The system refuses to truncate a file to which any process has a pending *vread*.

There is no primitive inverting *vread* to release the binding *vread* sets up so that the file may be closed. This can be only be done, clumsily, by reading another (plain) file onto the buffer area, or pulling the break back with *break*(2) to completely release the pages.

This call is peculiar to this version of UNIX. It will be superseded by more general virtual memory facilities in future versions of the system.

NAME

vswapon — add a swap device for interleaved paging/swapping

SYNOPSIS

vswapon (name)
char *name;

DESCRIPTION

Vswapon makes the argument block device available to the system for allocation for paging and swapping. The number of blocks to be made available, as well as the names of all potentially available devices are known to the system, and are present in the system configuration file (e.g. /usr/src/sys/conf/confhp.c).

SEE ALSO

swapon (8)

BUGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

This call is peculiar to this version of UNIX.

NAME

vtimes — get information about resource utilization

SYNOPSIS

vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;

DESCRIPTION

Vtimes returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file */usr/include/sys/vtimes.h:*

```
struct vtimes {
        int      vm_utime;              /* user time (*HZ) */
        int      vm_stime;              /* system time (*HZ) */
        /* divide next two by utime+stime to get averages */
        unsigned vm_idsrss;             /* integral of d+s rss */
        unsigned vm_ixrss;              /* integral of text rss */
        int      vm_maxrss;             /* maximum rss */
        int      vm_majflt;             /* major page faults */
        int      vm_minflt;             /* minor page faults */
        int      vm_nswap;              /* number of swaps */
        int      vm_inblk;              /* block reads */
        int      vm_oublk;              /* block writes */
};
```

The *vm_utime* and *vm_stime* fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The *vm_idrss* and *vm_ixrss* measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 512 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then *vm_idsrss* would have the value 5*60, where *vm_utime+vm_stime* would be the 60. *Vm_idsrss* integrates data and stack segment usage, while *vm_ixrss* integrates text segment usage. *Vm_maxrss* reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The *vm_majflt* field gives the number of page faults which resulted in disk activity; the *vm_minflt* field gives the number of page faults incurred in simulation of reference bits; *vm_nswap* is the number of swaps which occurred. The number of file system input/output events are reported in *vm_inblk* and *vm_oublk* These numbers account only for real i/o; data supplied by the caching mechanism is charged only to the first process to read or write the data.

SEE ALSO

time(2), wait3(2)

BUGS

This call is peculiar to this version of UNIX. The options and specifications of this system call are subject to change. It may be extended to include additional information in future versions of the system.

NAME

vwrite — write (virtually) to file

SYNOPSIS

vwrite(filedes, buffer, nbytes)
char *buffer;

DESCRIPTION

N.B.: This call is likely to be replaced by more general virtual memory facilities in the near future.

The *vwrite* system call is used in conjunction with *vread* to perform efficient updating of large files. After a call to *vread* and updating of the data in the buffer which was given to *vread,* a *vwrite* of the same buffer to the same *filedes* at the same offset in the file will cause data which has been modified since it was *vread* from (or *vwritten* to) the file to be returned to the file.

SEE ALSO

vread(2)

DIAGNOSTICS

Returns −1 on error: bad descriptor, buffer address, count or alignment as well as on physical I/O errors.

BUGS

The result of *vwrite* is defined only when no other *vread's* have occurred on *buffer* since the one matching the *vwrite.*

This call is peculiar to this version of UNIX. It will be superseded by more general virtual memory facilities in future versions of the system.

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 high byte of the word pointed to receives the low byte of the argument of *exit* when the child terminated. The low byte receives the termination status of the process. See *signal*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See *ptrace*(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

There is another entry *wait3*(2) which is provides additional options needed by the shell *csh*(1) to do job control.

SEE ALSO

wait3(2), exit(2), fork(2), signal(2)

DIAGNOSTICS

Returns −1 if there are no children not previously waited for.

ASSEMBLER (PDP-11)

(wait = 7.)
sys wait
(process ID in r0)
(status in r1)

The high byte of the status is the low byte of r0 in the child at termination.

NAME

wait3 — wait for process to terminate

SYNOPSIS

#include <wait.h>
#include <sys/vtimes.h>

wait3(status, options, vtimep)
union wait status;
int options;
struct vtimes *vtimep;

cc ... −ljobs

DESCRIPTION

The *status* and *option* words are described by definitions and macros in the file <wait.h>; the union and its bitfield definitions and associated macros given there provide convenient and mnemonic access to the word of status returned by a *wait3* call. See this file for more information.

There are two *options,* which may be combined by *or*ing them together. The first is WNOHANG which causes the *wait3* to not hang if there are no processes which wish to report status, rather returning a pid of 0 in this case as the result of the *wait3.* The second option is WUNTRACED which causes *wait3* to return information when children of the current process which are stopped but not traced (with *ptrace*(2)) because they received a SIGTTIN, SIGTTOU, SIGTSTP or SIGSTOP signal. See *sigsys*(2)) for a description of these signals.

The *vtimep* pointer is an optional structure where a *vtimes* structure is returned describing the resources used by the terminated process and all its children. This may be given as "0" if the information is not desired. Currently this information is not available for stopped processes.

SEE ALSO

wait(2), exit(2), fork(2), sigsys(2)

DIAGNOSTICS

Returns −1 if there are no children not previously waited for, or 0 if the WNOHANG option is given and there are no stopped or exited children.

BUGS

This call is peculiar to this version of UNIX. The options and specifications of this system call and even the call itself are subject to change. It may be replaced by other facilities in future versions of the system.

NAME

write — write on a file

SYNOPSIS

write(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe*(2) call.

Buffer is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 1024 characters long and begin on a 1024-byte boundary in the file are more efficient than any others.

Writes to the control terminal by a process which is not in the process group of the termainl and which is not ignoring or holding SIGTTOU signals cause the writer's process group to receive a SIGTTOU signal (See *jobs*(3) and the description of the LTOSTOP option in *tty*(4) for details).

On some systems *write* clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writeable set-user-id file owned by the super-user.

SEE ALSO

creat(2), open(2), pipe(2)

DIAGNOSTICS

Returns −1 on error: bad descriptor, buffer address, or count; physical I/O errors.

ASSEMBLER (PDP-11)

(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes
(byte count in r0)

BUGS

It would be nice to be able to call write and have the call return with an error indication if there was no buffer space for the written data, rather than blocking the process.

Processes which have been orphaned by their parents and have been inherited by *init*(8) never receive SIGTTOU signals. Output by such a process is permitted even when they are not in the process group of the control terminal.

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) and the Fortran compiler *f77*(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.

(3J) These functions are part of the job control facilities, contained in the library "**−ljobs**." The job control facilities are outlined in *jobs*(3).

(3M) These functions constitute the math library, *libm*. They are automatically loaded as needed by the Fortran compiler *f77*(1). 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)
/usr/lib/libjobs.a

SEE ALSO

stdio(3), nm(1), ld(1), cc(1), f77(1), intro(2)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* (see *intro*(2)) is set to the value EDOM or ERANGE. The values of EDOM and ERANGE are defined in the include file *<math.h>*.

ASSEMBLER (PDP-11)

In assembly language these functions may be accessed by simulating the C calling sequence. For example, *ecvt*(3) might be called this way:

```
setd
mov     $sign,-(sp)
mov     $decpt,-(sp)
mov     ndigit,-(sp)
movf    value,-(sp)
jsr     pc,_ecvt
add     $14.,sp
```

NAME

abort − generate a fault

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 be used for debugging.

SEE ALSO

adb(1), signal(2), exit(2)

DIAGNOSTICS

Usually 'IOT 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

assert − program verification

SYNOPSIS

#include <assert.h>

assert(expression)

DESCRIPTION

Assert is a macro that indicates *expression* is expected to be true at this point in the program. It causes an *exit*(2) with a diagnostic comment on the standard output when *expression* is false (0). Compiling with the *cc*(1) option **−DNDEBUG** effectively deletes *assert* from the program.

DIAGNOSTICS

'Assertion failed: file *f* line *n*.' *F* is the source file and *n* the source line number of the *assert* statement.

NAME

atof, atoi, atol — convert ASCII to numbers

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;

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)
```

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

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

dbminit, fetch, store, delete, firstkey, nextkey — data base subroutines

SYNOPSIS

```
typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **−ldbm**.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has '.dir' as its suffix. The second file contains all data and has '.pag' as its suffix.

Before a database can be accessed, it must be opened by *dbminit.* At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length '.dir' and '.pag' files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will traverse the data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr.*

BUGS

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older UNIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block.

Store will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

NAME

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;
char *buf;
```

DESCRIPTION

Ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by *ndigits*.

Gcvt converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf.* It attempts to produce *ndigit* significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3)

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 <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;

DESCRIPTION

Exp returns the exponential function of *x*.

Log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

Pow returns x^y.

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.

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 implemented as macros; they cannot be redeclared.

SEE ALSO

fopen(3), open(2)

NAME

fabs, floor, ceil — absolute value, floor, ceiling functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

Fabs returns the absolute value $|x|$.

Floor returns the largest integer not greater than *x*.

Ceil returns the smallest integer not less than *x*.

SEE ALSO

abs(3)

NAME

fopen, freopen, fdopen − open a stream

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)
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)
FILE *stream;

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

DESCRIPTION

Fread reads, into a block beginning at *ptr, nitems* of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

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.

BUGS

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 <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc*(3).

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on 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

gamma − log gamma function

SYNOPSIS

#include <math.h>

double gamma(x)
double x;

DESCRIPTION

Gamma returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer *signgam*. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0)
        error();
y = exp(y);
if(signgam)
        y = -y;
```

DIAGNOSTICS

A huge value is returned for negative integer arguments.

BUGS

There should be a positive indication of error.

NAME

getarg, iargc — command arguments to Fortran

SYNOPSIS

subroutine getarg(argno, string)
integer argno
character *(*) string

iargc()

DESCRIPTION

These procedures permit Fortran programs to access the command arguments. The integer function **iargc** returns the number of command arguments. The subroutine **getarg** stores the nth command argument in its second argument. The string is truncated or padded with blanks, in accord with the rules of Fortran character assignment.

The command

```
go arg1 argument2
```

will return 2 as the value of **iargc.** If **s** is declared character*4, then

```
call getarg(2, s)
```

will put "argu" in s.

SEE ALSO

exec(2)

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 VAX-11) 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 *value* if such a string is present, otherwise 0 (NULL).

SEE ALSO

environ(5), exec(2)

NAME

getfsent, getfsspec, getfsfile, setfsent, endfsent — get file system descriptor file entry

SYNOPSIS

```
#include <fstab.h>

struct fstab *getfsent()

struct fstab *getfsspec(name)
char *name;

struct fstab *getfsfile(name)
char *name;

int setfsend()

int endfsent()
```

DESCRIPTION

Getfsent, getfsspec and *getfsfile* each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, */usr/include/fstab.h.*

```
#define FSNMLG     16

struct fstab{
        char    fs_spec[FSNMLG];
        char    fs_file[FSNMLG];
        char    fs_type[3];
        int     fs_freq;
        int     fs_passno;
};
```

The fields have meanings described in *fstab*(5).

Getfsent reads the next line of the file, opening the file if necessary.

Setfsent opens and rewinds the file.

Endfsent closes the file.

Getfsspec and *getfsfile* sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until EOF is encountered.

FILES

/etc/fstab

SEE ALSO

fstab(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

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 { /* see getgrent(3) */
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group-ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

Getgrent simply reads the next line while *getgrgid* and *getgrnam* search until a matching *gid* or *name* is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3), getpwent(3), group(5)

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

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
Returns NULL (0) if name not 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)
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;
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 <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;

DESCRIPTION

Hypot and *cabs* return

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

exp(3) for *sqrt*

NAME

j0, j1, jn, y0, y1, yn – bessel functions

SYNOPSIS

```
#include &lt;math.h&gt;

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
double x;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause *y0, y1,* and *yn* to return a huge negative value and set *errno* to EDOM.

NAME

jobs – summary of job control facilities

SYNOPSIS

```
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/vtimes.h>
#include <wait.h>

int fildes, signo;
short pid, pgrp;
union wait status;
struct vtimes vt;

ioctl(fildes, TIOCSPGRP, &pgrp)
ioctl(fildes, TIOCGPGRP, &pgrp)

setpgrp(pid, pgrp)
getpgrp(pid)
killpg(pgrp, signo)

sigset(signo, action)
sighold(signo)
sigrelse(signo)
sigpause(signo)
sigsys(signo, action)

wait3(&status, options, &vt)

cc ... −ljobs
```

DESCRIPTION

The facilities described here are used to support the job control implemented in *csh*(1), and may be used in other programs to provide similar facilities. Because these facilities are not standard in UNIX and because the signal mechanisms are also slightly different, the associated routines are not in the standard C library, but rather in the **−ljobs** library.

For descriptions of the individual routines see the various sections listed in SEE ALSO below. This section attempt only to place these facilities in context, not to explain the semantics of the individual calls.

Terminal arbitration mechanisms.

The job control mechanism works by associating with each process a number called a *process group* ; related processes (e.g. in a pipeline) are given the same process group. The system assigns a single process group number to each terminal. Processes running on a terminal are given read access to that terminal only if they are in the same process group as that terminal.

Thus a command interpreter may start several jobs running in different process groups and arbitrate access to the terminal by controlling which, if any, of these processes is in the same process group as the terminal. When a process which is not in the process group of the terminal tries to read from the terminal, all members of the process group of the process receive a SIGTTIN signal, which normally then causes them to stop until they are continued with a SIGCONT signal. (See *sigsys*(2) for a description of these signals; *tty*(4) for a description of process groups.)

If a process which is not in the process group of the terminal attempts to change the terminals mode, the process group of that process is sent a SIGTTOU signal, causing the process group to stop. A similar mechanism is (optionally) available for output, causing processes to block with SIGTTOU when they attempt to write to the terminal while not in its process group; this is controlled by the LTOSTOP bit in the tty mode word, enabled by "stty tostop" and disabled (the default) by "stty −tostop." (The LTOSTOP bit is described in *tty*(4)).

How the shell manipulates process groups.

A shell which is interactive first establishes its own process group and a process group for the terminal; this prevents other processes from being inadvertantly stopped while the terminal is under its control. The shell then assigns each job it creates a distinct process group. When a job is to be run in the foreground, the shell gives the terminal to the process group of the job using the TIOCSPGRP ioctl (See *ioctl*(2) and *tty*(4)). When a job stops or completes, the shell reclaims the terminal by resetting the terminals process group to that of the shell using TIOCSPGRP again.

Shells which are running shell scripts or running non-interactively do not manipulate process groups of jobs they create. Instead, they leave the process group of sub-processes and the terminal unchanged. This assures that if any sub-process they create blocks for terminal i/o, the shell and all its sub-processes will be blocked (since they are a single process group). The first interactive parent of the non-interactive shell can then be used to deal with the stoppage.

Processes which are orphans (whose parents have exited), and descendants of these processes are protected by the system from stopping, since there can be no interactive parent. Rather than blocking, reads from the control terminal return end-of-file and writes to the control terminal are permitted (i.e. LTOSTOP has no effect for these processes.) Similarly processes which ignore or hold the SIGTTIN or SIGTTOU signal are not sent these signals when accessing their control terminal; if they are not in the process group of the control terminal reads simply return end-of-file. Output and mode setting are also allowed.

Before a shell *suspends* itself, it places itself back in the process group in which it was created, and then sends this original group a stopping signal, stopping the shell and any other intermediate processes back to an interactive parent. The shell also restores the process group of the terminal when it finishes, as the process which then resumes would not necessarily be in control of the terminal otherwise.

Naive processes.

A process which does not alter the state of the terminal, and which does no job control can invoke subprocesses normally without worry. If such a process issues a *system*(3) call and this command is then stopped, both of the processes will stop together. Thus simple processes need not worry about job control, even if they have "shell escapes" or invoke other processes.

Processes which modify the terminal state.

When first setting the terminal into an unusual mode, the process should check, with the stopping signals held, that it is in the foreground. It should then change the state of the terminal, and set the catches for SIGTTIN, SIGTTOU and SIGTSTP. The following is a sample of the code that will be needed, assuming that unit 2 is known to be a terminal.

```
	short	tpgrp;
	...

retry:
	sigset(SIGTSTP, SIG_HOLD);
	sigset(SIGTTIN, SIG_HOLD);
	sigset(SIGTTOU, SIG_HOLD);
	if (ioctl(2, TIOCGPGRP, &tpgrp) != 0)
		goto nottty;
	if (tpgrp != getpgrp(0)) { /* not in foreground */
		sigset(SIGTTOU, SIG_DFL);
		kill(0, SIGTTOU);
		/* job stops here waiting for SIGCONT */
```

```
		goto retry;
	}
	...save old terminal modes and set new modes...
	sigset(SIGTTIN, onstop);
	sigset(SIGTTOU, onstop);
	sigset(SIGTSTP, onstop);
```

It is necessary to ignore SIGTSTP in this code because otherwise our process could be moved from the foreground to the background in the middle of checking if it is in the foreground. The process holds all the stopping signals in this critical section so no other process in our process group can mess us up by blocking us on one of these signals in the middle of our check. (This code assumes that the command interpreter will not move a process from foreground to background without stopping it; if it did we would have no way of making the check correctly.)

The routine which handles the signal should clear the catch for the stop signal and *kill*(2) the processes in its process group with the same signal. The statement after this *kill* will be executed when the process is later continued with SIGCONT.

Thus the code for the catch routine might look like:

```
	...
	sigset(SIGTSTP, onstop);
	sigset(SIGTTIN, onstop);
	sigset(SIGTTOU, onstop);
	...

onstop(signo)
	int signo;
{
	... restore old terminal state ...
	sigset(signo, SIG_DFL);
	kill(0, signo);
	/* stop here until continued */
	sigset(signo, onstop);
	... restore our special terminal state ...
}
```

This routine can also be used to simulate a stop signal.

If a process does not need to save and restore state when it is stopped, but wishes to be notified when it is continued after a stop it can catch the SIGCONT signal; the SIGCONT handler will be run when the process is continued.

Processes which lock data bases such as the password file should ignore SIGTTIN, SIGTTOU, and SIGTSTP signals while the data bases are being manipulated. While a process is ignoring SIGTTIN signals, reads which would normally have hung will return end-of-file; writes which would normally have caused SIGTTOU signals are instead permitted while SIGTTOU is ignored.

Interrupt-level process handling.

Using the mechanisms of *sigset*(3) it is possible to handle process state changes as they occur by providing an interrupt-handling routine for the SIGCHLD signal which occurs whenever the status of a child process changes. A signal handler for this signal is established by:

```
	sigset(SIGCHLD, onchild);
```

The shell or other process would then await a change in child status with code of the form:

```
recheck:
        sighold(SIGCHLD);           /* start critical section */
        if (no children to process) {
                sigpause(SIGCHLD);  /* release SIGCHLD and pause */
                goto recheck;
        }
        sigrelse(SIGCHLD);          /* end critical region */
        /* now have a child to process */
```

Here we are using *sighold* to temporarily block the SIGCHLD signal during the checking of the data structures telling us whether we have a child to process. If we didn't block the signal we would have a race condition since the signal might corrupt our decision by arriving shortly after we had finished checking the condition but before we paused.

If we need to wait for something to happen, we call *sigpause* which automically releases the hold on the SIGCHLD signal and waits for a signal to occur by starting a *pause*(2). Otherwise we simply release the SIGCHLD signal and process the child. *Sigpause* is similar to the PDP-11 *wait* instruction, which returns the priority of the processor to the base level and idles waiting for an interrupt.

It is important to note that the long-standing bug in the signal mechanism which would have lost a SIGCHLD signal which occurred while the signal was blocked has been fixed. This is because *sighold* uses the SIG_HOLD signal set of *sigsys*(2) to prevent the signal action from being taken without losing the signal if it occurs. Similarly, a signal action set with *sigset* has the signal held while the action routine is running, much as a the interrupt priority of the processor is raised when a device interrupt is taken.

In this interrupt driven style of termination processing it is necessary that the *wait* calls used to retrieve status in the SIGCHLD signal handler not block. This is because a single invocation of the SIGCHLD handler may indicate an arbitrary number of process status changes: signals are not queued. This is similar to the case in a disk driver where several drives on a single controller may report status at once, while there is only one interrupt taken. It is even possible for no children to be ready to report status when the SIGCHLD handler is invoked, if the signal was posted while the SIGCHLD handler was active, and the child was noticed due to a SIGCHLD initially sent for another process. This causes no problem, since the handler will be called whenever there is work to do; the handler just has to collect all information by calling *wait3* until it says no more information is available. Further status changes are guaranteed to be reflected in another SIGCHLD handler call.

Restarting system calls.

In older versions of UNIX "slow" system calls were interrupted when signals occurred, returning EINTR. The new signal mechanism *sigset*(3) normally restarts such calls rather than interrupting them. To summarize: *pause* and *wait* return error EINTR (as before), *ioctl* and *wait3* restart, and *read* and *write* restart unless some data was read or written in which case they return indicating how much data was read or written. In programs which use the older *signal*(2) mechanisms, all of these calls return EINTR if a signal occurs during the call.

SEE ALSO

csh(1), ioctl(2), killpg(2), setpgrp(2), sigsys(2), wait3(2), signal(3), tty(4)

BUGS

The job control facilities are not available in standard version 7 UNIX. These facilities are still under development and may change in future releases of the system as better inter-process communication facilities and support for virtual terminals become available. The options and specifications of these system calls and even the calls themselves are thus subject to change.

NAME

l3tol, ltol3 — convert between 3-byte integers and long integers

SYNOPSIS

l3tol(lp, cp, n)
long *lp;
char *cp;

ltol3(cp, lp, n)
char *cp;
long *lp;

DESCRIPTION

L3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the i-numbers are three bytes long.

SEE ALSO

filsys(5)

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[];

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 <nlist.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 **/vmunix**. 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.

BUGS

On other versions of UNIX you must include <a.out.h> rather than <nlist.h>; this is unfortunate, but *<a.out.h>* can't be used on the VAX because it contains a **union** which can't be initialized.

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

plot: openpl et al. — graphics interface

SYNOPSIS

openpl()

erase()

label(s)
char s[];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

move(x, y)

cont(x, y)

point(x, y)

linemod(s)
char s[];

space(x0, y0, x1, y1)

closepl()

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot*(5) for a description of their effect. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

String arguments to *label* and *linemod* are null-terminated, and do not contain newlines.

Various flavors of these functions exist for different output devices. They are obtained by the following *ld*(1) options:

−lplot device-independent graphics stream on standard output for *plot*(1) filters
−l300 GSI 300 terminal
−l300s GSI 300S terminal
−l450 DASI 450 terminal
−l4014 Tektronix 4014 terminal

SEE ALSO

plot(5), plot(1), graph(1)

NAME

popen, pclose − initiate I/O to/from a process

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 <stdio.h>

printf(format [, arg] ...)
char *format;

fprintf(stream, format [, arg] ...)
FILE *stream;
char *format;

sprintf(s, format [, arg] ...)
char *s, format;

DESCRIPTION

Printf places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places 'output' in the string *s*, followed by the character '\0'.

Each of these functions converts, formats, and prints its arguments after the 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.

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 through MAXUINT, where MAXUINT equals 4294967295 on a VAX-11 and 65536 on a PDP-11).

% 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 π 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)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;

DESCRIPTION

Putc appends the character *c* to the named output *stream*. It returns the character written.

Putchar(c) is defined as *putc(c, stdout).*

Fputc behaves like *putc,* but is a genuine function rather than a macro. It may be used to save on object text.

Putw appends word (i.e. **int**) *w* to the output *stream*. It returns the word written. *Putw* neither assumes nor causes special alignment in the file.

The standard stream *stdout* is normally buffered if and only if the output does not refer to a terminal; this default may be changed by *setbuf*(3). The standard stream *stderr* is by default unbuffered unconditionally, but use of *freopen* (see *fopen*(3)) will cause it to become buffered; *setbuf,* again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. *Fflush* (see *fclose*(3)) may be used to force the block out early.

SEE ALSO

fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, *ferror*(3) should be used to detect *putw* errors.

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) for *fwrite*

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 (*compar)();

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1)

NAME

rand, srand − random number generator

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

re_comp, re_exec — regular expression handler

SYNOPSIS

char *re_comp(s)
char *s;

re_exec(s)
char *s;

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. *Re_exec* checks the argument string against the last string passed to *re_comp.*

Re_comp returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If *re_comp* is passed 0 or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both *re_comp* and *re_exec* may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for *ed* (1), given the above difference.

SEE ALSO

ed(1), ex(1)

DIAGNOSTICS

Re_exec returns -1 for an internal error.

Re_comp returns one of the following strings if an error occurs: "No previous regular expression", "Regular expression too long", "unmatched \(", "missing]", "too many \(\) pairs", "unmatched \)".

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.

e
f a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by

an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o** and **x** may be capitalized or preceeded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e** or **f** may be capitalized or preceded by **l** to indicate a pointer to **double** rather than to **float**. The conversion characters **d**, **o** and **x** may be preceeded by **h** to indicate a pointer to **short** rather than to **int**.

The *scanf* functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E−1   thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain *'thompson\0'*. Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip '0123', and place the string '56\0' in *name*. The next call to *getchar* will return 'a'.

SEE ALSO

atof(3), getc(3), printf(3)

DIAGNOSTICS

The *scanf* functions return **EOF** on end of input, and a short count for missing or illegal data items.

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;

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

Longjmp restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the function that invoked *setjmp*, which must not itself have returned in the interim. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2)

NAME

sigset, signal, sighold, sigignore, sigrelse, sigpause — manage signals

SYNOPSIS

#include <signal.h>
void action();
int sig;

sigset(sig, action)
signal(sig, action)

sighold(sig)
sigignore(sig)
sigrelse(sig)

sigpause(sig)

cc ... −ljobs

DESCRIPTION

This is a package of signal management functions to manage the signals as described in *sigsys*(2). These functions are available only in this version of UNIX, and should not be used when the mechanisms of *signal*(2) would suffice, as they would then impair portability. These functions are contained in the *jobs* library, obtained by specifying the loader option **−ljobs**.

Sigset is used to provide a default signal handler for signal *sig*. This function is remembered across subsequent calls to the other functions, and need not be specified again. After *sigset* instances of *sig* will cause an interrupt to be taken at *func*, with the signal then held so that recursive trapping due to the signal will not occur. During normal return from *func*, the routines arrange for the signal action to be restored so that subsequent signals will also trap to *func*. If a non-local exit is to be taken, then *sigrelse* must be called to un-hold the signal action, restoring the original catch. *Func* may also be specified as SIG_DFL, SIG_IGN or SIG_HOLD, as described in *sigsys*(2). The value specified on the previous call to *sigset* is returned; if *sigset* has never been called, then the default action inherited from the system is returned.

Signal is like *sigset*, but the signal will not be held when the action routine is called; rather it will have reverted to SIG_DFL. This is generally unsafe, but is included for backwards compatibility to the old signal mechanism. It should not be used.

Sighold and *sigrelse* may be used to block off *sig* in a piece of code where it cannot be tolerated. After *sigrelse* the catch initially set with *sigset* will be restored.

Sigignore can be used to temporarily set the action for *sig* to ignore the signal. If the signal had been held before the call to *sigignore*, any pending instance of the signal will be discarded.

Sigpause may be used by a routine which wishes to check for some condition produced at interrupt level by the *sig* signal, and then to pause waiting for the condition to arise with the catch of the signal enabled. In correct usage it must be preceded by an instance of *sighold* to block the signal. *Sigpause* is like *pause* in that it will return after *any* signal is processed. The usual thing to do then is to reenable the hold with *sighold*, check the condition again, and *sigpause* again if the condition has not arisen.

SEE ALSO

sigsys(2), signal(2), jobs(3), tty(4)

BUGS

Sighold and *sigrelse* do not nest; the first *sigrelse* restores the default catch.

These functions store information in data space. You thus must call *sigsys*(2) rather than any of *sigset* or *signal* after a *vfork*(2) in the child which is to then *exec*(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 π.

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 π.

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 <math.h>

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

The functions described in Sections 3S constitute an efficient user-level buffering scheme. The in-line macros *getc* and *putc*(3) handle characters quickly. The higher level routines *gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf, fwrite* all use *getc* and *putc;* they can be freely intermixed.

A file with associated buffering is called a *stream,* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

stdin standard input file
stdout standard output file
stderr standard error file

A constant 'pointer' NULL (0) designates no stream at all.

An integer constant EOF (−1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file <stdio.h> of pertinent macro definitions. The functions and constants mentioned in sections labeled 3S are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: *getc, getchar, putc, putchar, feof, ferror, fileno.*

SEE ALSO

open(2), close(2), read(2), write(2)

DIAGNOSTICS

The value EOF is returned uniformly to indicate that a **FILE** pointer has not been initialized with *fopen,* input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default and attempts to do this transparently by flushing the output whenever a *read*(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard i/o routines but use *read*(2) themselves to read from the standard input.

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;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, 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*. *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.

BUGS

Strcmp uses native character comparison, which is signed on PDP11's and VAX-11's, unsigned on other machines.

NAME

swab — swap bytes

SYNOPSIS

swab(from, to, nbytes)
char *from, *to;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP11's and 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.

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;

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	−ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

AUTHOR

William Joy

BUGS

NAME

ttyname, isatty, ttyslot — find name of a terminal

SYNOPSIS

char *ttyname(fildes)

isatty(fildes)

ttyslot()

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

Ttyslot returns the number of the entry in the *ttys*(5) file for the control terminal of the current process.

FILES

/dev/*
/etc/ttys

SEE ALSO

ioctl(2), ttys(5)

DIAGNOSTICS

Ttyname returns a null pointer (0) if *fildes* does not describe a terminal device in directory '/dev'.

Ttyslot returns 0 if '/etc/ttys' is inaccessible or if it cannot determine the control terminal.

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)
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

valloc — aligned memory allocator

SYNOPSIS

char *valloc(size)
unsigned size;

DESCRIPTION

Valloc allocates *size* bytes aligned on a boundary adequate for *vread*(2)). It is implemented by calling *malloc*(3) with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

DIAGNOSTICS

Valloc returns 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.

BUGS

Vfree isn't implemented.

NAME

varargs — variable argument list

SYNOPSIS

#include <varargs.h>
function(**va_alist**)
va_dcl
va_list *pvar*;
va_start(*pvar*);
f = **va_arg**(*pvar*, *type*);
va_end(*pvar*);

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines which have variable argument lists (such as *printf(3))* that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for **va_alist**. Note that there is no semicolon after **va_dcl.**

va_list is a type which can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

va_start (pvar) is called to initialize *pvar* to the beginning of the list.

va_arg (*pvar,type*) will return the next argument in the list pointed to by *pvar*. *Type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end (*pvar*) is used to finish up.

Multiple traversals, each bracketted by **va_start** .. **va_end**, are possible.

EXAMPLE

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[100];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
        return execv(file, args);
}
```

BUGS

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, *execl* passes a 0 to signal the end of the list. *Printf* can tell how many arguments are there by the format.

NAME

intro − introduction to special files

DESCRIPTION

This section describes the special files and related driver functions available on the system. In this section the SYNOPSIS section gives a sample specification of the related drivers for use in a system description to the *config*(8) program. The DIAGNOSTICS section lists messages which may appear on the console and in the system error log /usr/adm/messages due to errors in device operation.

SEE ALSO

config(8)

NAME

autoconf — diagnostics from autoconfiguration code

DESCRIPTION

When UNIX bootstraps it probes the innards of the machine it is running on and locates controllers, drives, and other devices, printing out what it finds on the console. This procedure is driven by a system configuration table which is processed by *config*(8) and compiled into each kernel.

Devices in NEXUS slots are normally noted, thus memory controllers, UNIBUS and MASSBUS adaptors. Devices which are not supported which are found in NEXUS slots are noted also.

MASSBUS devices are located by a very deterministic procedure since MASSBUS space is completely probable very easily. If devices exist which are not configured they will be silently ignored; if devices exist of unsupported type they will be noted.

UNIBUS devices are located by probing to see if their control-status registers respond. If not, they are silently ignored. If the control status register responds but the device cannot be made to interrupt, a diagnostic warning will be printed on the console and the device will not be available to the system. (A command *attach*(8) is planned to cause the device to be attached irregardless of its failure to interrupt, after the system is bootstrapped, for irksome devices. This is not in as of this writing, however.)

A generic system may be built which picks its root device at boot time as the "best" available device (MASSBUS disks are better than SMD UNIBUS disks are better than RK07's; the device must be drive 0 to be considered.) If such a system is booted with the RB_ASKNAME option of (see *reboot*(2V)), then the name of the root device is read from the console terminal at boot time, and any available device may be used.

SEE ALSO

config(8)

DIAGNOSTICS

cpu type %d not configured. You tried to boot UNIX on a cpu type which it doesn't (or at least this compiled version of UNIX doesn't) understand.

mba%d at tr%d. A MASSBUS adapter was found in tr%d (the NEXUS slot number). UNIX will call it mba%d.

%d mba's not configured. More MASSBUS adapters were found on the machine than were declared in the machine configuration; the excess MASSBUS adapters will not be accessible.

uba%d at tr%d. A UNIBUS adapter was found in tr%d (the NEXUS slot number). UNIX will call it uba%d.

dr32 unsupported (at tr %d). A DR32 interface was found in a NEXUS, for which UNIX does not have a driver.

mcr%d at tr%d. A memory controller was found in tr%d (the NEXUS slot number). UNIX will call it mcr%d.

5 mcr's unsupported. UNIX supports only 4 memory controllers per cpu.

mpm unsupported (at tr%d). Multi-port memory is unsupported in the sense that UNIX does not know how to poll it for ECC errors.

%s%d at mba%d drive %d. A tape formatter or a disk was found on the MASSBUS; for disks %s%d will look like "hp0", for tape formatters like "ht1" or "mt1". The drive number comes from the unit plug on the drive or in the TM formatter (**not** on the tape drive; see below).

%s%d at %s%d slave %d. (For MASSBUS devices). Which would look like "tu0 at ht0 slave 0" or "mu0 at mt0 slave 0", where **tu0** or **mu0** is the name for the tape device and **ht0** or **mt0**

is the name for the formatter. A tape slave was found on the tape formatter at the indicated drive number (on the front of the tape drive). UNIX will call the device, e.g., **tu0** or **mu0**.

%s%d at uba%d csr %o vec %o ipl %x. The device %s%d, e.g. dz0 was found on uba%d at control-status register address %o and with device vector %o. The device interrupted at priority level %x.

%s%d at uba%d csr %o zero vector. The device did not present a valid interrupt vector, rather presented 0 (a passive release condition) to the adapter.

%s%d at uba%d csr %o didn't interrupt. The device did not interrupt, likely because it is broken, hung, or not the kind of device it is advertised to be.

%s%d at %s%d slave %d. (For UNIBUS devices). Which would look like "up0 at sc0 slave 0", where **up0** is the name of a disk drive and **sc0** is the name of the controller. Analogous to MASSBUS case.

BUGS

Should write *attach*(8) and system call it needs to work.

NAME

bk — line discipline for machine-machine communication

SYNOPSIS

pseudo-device bk

DESCRIPTION

This line discipline provides a replacement for the old and new tty drivers described in *tty*(4) when high speed output to and especially input from another machine is to be transmitted over a asynchronous communications line. The discipline was designed for use by the Berkeley network *net*(1). It may be suitable for uploading of data from microprocessors into the system. If you are going to send data over asynchronous communications lines at high speed into the system, you must use this discipline, as the system otherwise may detect high input data rates on terminal lines and disables the lines; in any case the processing of such data when normal terminal mechanisms are involved saturates the system.

The line discipline is enabled by a sequence:

```
#include <sgtty.h>
int ldisc = NETLDISC, fildes; ...
ioctl(fildes, TIOCSETD, &ldisc);
```

A typical application program then reads a sequence of lines from the terminal port, checking header and sequencing information on each line and acknowledging receipt of each line to the sender, who then transmits another line of data. Typically several hundred bytes of data and a smaller amount of control information will be received on each handshake.

The old standard teletype discipline can be restored by doing:

```
ldisc = OTTYDISC;
ioctl(fildes, TIOCSETD, &ldisc);
```

While in networked mode, normal teletype output functions take place. Thus, if an 8 bit output data path is desired, it is necessary to prepare the output line by putting it into RAW mode using *ioctl*(2). This must be done **before** changing the discipline with TIOCSETD, as most *ioctl*(2) calls are disabled while in network line-discipline mode.

When in network mode, input processing is very limited to reduce overhead. Currently the input path is only 7 bits wide, with newline the only recognized character, terminating an input record. Each input record must be read and acknowledged before the next input is read as the system refuses to accept any new data when there is a record in the buffer. The buffer is limited in length, but the system guarantees to always be willing to accept input resulting in 512 data characters and then the terminating newline.

User level programs should provide sequencing and checksums on the information to guarantee accurate data transfer.

SEE ALSO

tty(4)

DIAGNOSTICS

None.

BUGS

The Purdue uploading line discipline, which provides 8 bits and uses timeout's to terminate uploading should be incorporated into the standard system, as it is much more suitable for microprocessor connections.

Inclusion of this line discipline causes the system to use the input silos on dz's and dh's. This causes problems with some terminals, which require ^S/^Q handshaking to operate but have inadequate buffering to tolerate even a small number of characters transmitted after they send a

^S. In particular this problem existed on early VT100's (where, however, an ECO exists to fix this problem.)

NAME

cons − VAX-11 console interface

DESCRIPTION

The console is available to the processor through the console registers. It acts like a normal terminal, except that when the local functions are not disabled, control-P puts the console in local console mode (where the prompt is ">>>"). The operation of the console in this mode varies slightly per-processor.

On an 11/780 you can return to the conversational mode using the command "se t p" if the processor is still running or "continue" if it is halted. The latter command may be abbreviated "c". If you hit the break key on the console, then the console will go into ODT (console debugger mode). Hit a "P" (upper-case letter p) to get out of this mode.

On an 11/750 the processor is halted whenever the console is not in conversational mode, and typing "C" returns to conversational mode. When in console mode on an 11/750 which has a remote diagnosis module, a ^D will put you in remote diagnosis mode, where the prompt will be "RDM>". The command "ret" will return from remote diagnosis mode to local console mode.

With the above proviso's the console works like any other UNIX terminal.

FILES

/dev/console

SEE ALSO

tty(4), reboot(8)
VAX Hardware Handbook

DIAGNOSTICS

None.

NAME

ct — phototypesetter interface

SYNOPSIS

device ct0 at uba0 csr 0167760 vector ctintr

DESCRIPTION

This provides the interfact to a Graphic Systems C/A/T phototypesetter. Bytes written on the file specify font, size, and other control information as well as the characters to be flashed. The coding will not be described here.

Only one process may have this file open at a time. It is write-only.

FILES

/dev/cat

SEE ALSO

troff(1)
Phototypesetter interface specification

DIAGNOSTICS

None.

NAME

dh, dm − DH-11/DM-11 communications multiplexer

SYNOPSIS

device dh0 at uba0 csr 0160020 vector dhrint dhxint
device dm0 at uba0 csr 0170500 vector dmintr

DESCRIPTION

A dh-11 provides 16 communication lines; dm-11's may be optionally paired with dh-11's to provide modem control for the lines.

Each line attached to the DH-11 communications multiplexer behaves as described in *tty*(4). Input and output for each line may independently be set to run at any of 16 speeds; see *tty*(4) for the encoding.

Bit *i* of flags may be specified for a dh to say that a line is not properly connected, and that the line should be treated as hard-wired with carrier always present. Thus specifying "flags 0x0004" in the specification of dh0 would cause line ttyh2 to be treated in this way.

If the Berknet line driver *bk*(4) is included in the system, then the dh driver will use its input silos and poll for input at each clock tick (normally 1/50'th or 1/60'th of a second) rather than taking an interrupt on each input character.

FILES

/dev/tty[hi][0-9a-f]
/dev/ttyd[0-9a-f]

SEE ALSO

tty(4)

DIAGNOSTICS

dh%d: NXM. No response from UNIBUS on a dma transfer within a timeout period. This is often followed by a UNIBUS adapter error. This occurs most frequently when the UNIBUS is heavily loaded and when devices which hog the bus (such as rk07's) are present. It is not serious.

dh%d: silo overflow. The 64 character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system will then print a message on the console with interrupts disabled. If the Berknet *net*(1) is running on a *dh* line at high speed (e.g. 9600 baud), there is only 1/15th of a second of buffering capacity in the silo, and overrun is possible. This may cause a few input characters to be lost to users and a network packet is likely to be corrupted, but the network will recover. It is not serious.

NAME

dn − dn-11 automatic calling unit interface

SYNOPSIS

device dn0 at uba0 csr 0175200 vector dnintr

DESCRIPTION

A dn-11 is a controller for an automatic calling unit (ACU). With a dn-11, a call unit, and a modem a computer may dial a telephone number and establish a data link. Hardware supporting this interface includes the DEC DN11 and Able Quadracall controllers working with either Bell 801 or Racal/Vadic 801 or 811 call units.

A dn-11 interface can control as many as four call units. In normal operation a data link is established by opening the file associated with a call unit, sending the desired phone number in a single *write*(2) operation, then opening the terminal line attached to the appropriate outgoing modem. If the phone call was successful and the remote host presented a carrier signal, the open on the terminal line should be successful. At this point the call unit should be closed and a link has been established. This sequence is normally performed for the user by one of *cu*(1), *uucp*(1), or *dnd*(8).

Each controller/call unit pair has different programming characteristics which impact the programmer; the dn-11 interface presents a *raw interface* rather than imbed local dependencies in the device driver. For example, the Racal/Vadic 811 call unit is multiplexed, providing the user with the ability to place phones calls on as many as 64 separate phone lines from a single interface. This multiplexing is carried out based on the first two digits of the phone number presented by the user (the first digit selects the modem's rack and modem type, while the second digit selects the specific modem in the rack).

The legal dn-11 dial codes are: **0-9**, ***** (: is a synonym), **#** (; is a synonym), **=** (**w** is a synonym), and **−**. A phone number may optionally be terminated with a **<** or **e**. Phone numbers may be no more than 40 characters long.

By convention /dev/cu[al]0 are connected to a 300 baud interface, while /dev/cu[al]1 provides a 1200 baud interface.

FILES

/dev/cua[0-9]	ACU interfaces
/dev/cul[0-9]	associated outgoing terminal line

SEE ALSO

rv(4)

NAME

drum − paging device

DESCRIPTION

This file refers to the paging device in use by the system. This may actually be a subdevice of one of the disk drivers, but in a system with paging interleaved across multiple disk drives it provides an indirect driver for the multiple drives.

FILES

/dev/drum

BUGS

Reads from the drum are not allowed across the interleaving boundaries. Since these only occur every .5Mbytes or so, and since the system never allocates blocks across the boundary, this is usually not a problem.

NAME

dz — DZ-11 communications multiplexer

SYNOPSIS

device dz0 at uba0 csr 0160100 vector dzrint dzxint

DESCRIPTION

A dz-11 provides 8 communication lines with partial modem control, adequate for UNIX dialup use. Each line attached to the DZ-11 communications multiplexer behaves as described in *tty*(4) and may be set to run at any of 16 speeds; see *tty*(4) for the encoding.

Bit *i* of flags may be specified for a dz to say that a line is not properly connected, and that the line should be treated as hard-wired with carrier always present. Thus specifying "flags 0x04" in the specification of dz0 would cause line tty02 to be treated in this way.

The dz driver normally uses its input silos and polls for input at each clock tick (1/60'th of a second) rather than taking an interrupt on each input character.

FILES

/dev/tty[0-9][0-9]
/dev/ttyd[0-9a-f]   dialups

SEE ALSO

tty(4)

DIAGNOSTICS

dz%d: silo overflow. The 64 character input silo overflowed before it could be serviced. This can happen if a hard error occurs when the CPU is running with elevated priority, as the system will then print a message on the console with interrupts disabled. If the Berknet *net*(1) is running on a *dz* line at high speed (e.g. 9600 baud), there is only 1/15th of a second of buffering capacity in the silo, and overrun is possible. This may cause a few input characters to be lost to users and a network packet is likely to be corrupted, but the network will recover. It is not serious.

NAME

fl − floppy interface

DESCRIPTION

This is a simple interface to the D.E.C. RX01 floppy disk unit, which is part of the console LSI-11 subsytem for VAX-11/780's. Access is given to the entire floppy consisting of 77 tracks of 26 sectors of 128 bytes.

All i/o is raw; the seek addresses in raw transfers should be a multiple of 128 bytes and a multiple of 128 bytes should be transferred, as in other "raw" disk interfaces.

FILES

/dev/floppy

SEE ALSO

arff(8)

DIAGNOSTICS

None.

BUGS

Multiple console floppies are not supported.

If a write is given with a count not a multiple of 128 bytes then the trailing portion of the last sector will be zeroed.

NAME

hk – RK6-11/RK06 and RK07 moving head disk

SYNOPSIS

controller hk0 at uba? csr 0177440 vector rkintr
disk rk0 at hk0 drive 0
disk rk1 at hk0 drive 1

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, etc. The standard device names begin with "hk" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The origin and size of the pseudo-disks on each drive are as follows:

RK07 partitions:

disk	start	length	cyl
hk?a	0	15884	0-240
hk?b	15906	10032	241-392
hk?c	0	53790	0-814
hk?g	26004	27786	393-813

RK06 partitions

disk	start	length	cyl
hk?a	0	15884	0-240
hk?b	15906	11154	241-409
hk?c	0	27126	0-410

On a dual RK-07 system partition hk?a is used for the root for one drive and partition hk?g for the /usr file system. If large jobs are to be run using hk?b on both drives as swap area provides a 10Mbyte paging area. Otherwise partition hk?c on the other drive is used as a single large file system.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/hk[0-7][a-h]	block files
/dev/rhk[0-7][a-h]	raw files

SEE ALSO

hp(4), up(4)

DIAGNOSTICS

rk%d%c: hard error sn%d cs2=%b ds=%b er=%b. An unrecoverable error occured during transfer of the specified sector of the specified disk partition. The contents of the cs2, ds and er registers are printed in octal and symbolically with bits decoded. The error was either unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover the error.

rk%d: write locked. The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

rk%d: not ready. The drive was spun down or off line when it was accessed. The i/o operation is not recoverable.

rk%d: not ready (came back!). The drive was not ready, but after printing the message about being not ready (which takes a fraction of a second) was ready. The operation is recovered if no further errors occur.

rk%d%c: soft ecc sn%d. A recoverable ECC error occurred on the specified sector of the specified disk partition. This happens normally a few times a week. If it happens more frequently than this the sectors where the errors are occuring should be checked to see if certain cylinders on the pack, spots on the carriage of the drive or heads are indicated.

hk%d: lost interrupt. A timer watching the controller detected no interrupt for an extended period while an operation was outstanding. This indicates a hardware or software failure. There is currently a hardware/software problem with spinning down drives while they are being accessed which causes this error to occur. The error causes a UNIBUS reset, and retry of the pending operations. If the controller continues to lose interrupts, this error will recur a few seconds later.

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

DEC-standard error logging should be supported.

A program to analyze the logged error information (even in its present reduced form) is needed.

The partition tables for the file systems should be read off of each pack, as they are never quite what any single installation would prefer, and this would make packs more portable.

NAME

hp − RP06, RM03, RM05, RM80, RP07 MASSBUS moving-head disk

SYNOPSIS

disk hp0 at mba0 drive 0

DESCRIPTION

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, etc. The standard device names begin with "hp" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The origin and size of the pseudo-disks on each drive are as follows:

RP06 partitions

disk	start	length	cyls
hp?a	0	15884	0-37
hp?b	15884	33440	38-117
hp?c	0	340670	0-814
hp?g	49324	291280	118-814

RM03 partitions

disk	start	length	cyls
hp?a	0	15884	0-99
hp?b	16000	33440	100-309
hp?c	0	131680	0-822
hp?g	49600	81984	310-822

RM05 partitions

disk	start	length	cyls
hp?a	0	15884	0-26
hp?b	16416	33440	27-81
hp?c	0	500384	0-822
hp?d	341696	15884	562-588
hp?e	358112	55936	589-680
hp?f	414048	86240	681-822
hp?g	341696	158592	562-822
hp?h	49856	291346	82-561

RM80 partitions

disk	start	length	cyls
hp?a	0	15884	0-36
hp?b	16058	33440	37-114
hp?c	0	242606	0-558
hp?g	49910	82080	115-304
hp?h	132370	110143	305-558

RP07 partitions

disk	start	length	cyls
hp?a	0	15884	0-9
hp?b	16000	64000	10-49
hp?c	0	1008000	0-629
hp?d	528000	15884	330-339
hp?e	544000	258000	340-499
hp?f	800000	207850	500-629
hp?g	528000	479850	330-629
hp?h	80000	448000	50-329

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter. The hp?a partition is normally used for the root file system, the hp?b partition as a paging area, and the hp?c partition for pack-pack copying (it maps the entire disk). On rp06's and rm03's the hp?g partition maps the rest of the pack. On rm80's, rm05's and rp07's, both hp?g and hp?h are used to map the remaining cylinders.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/hp[0-7][a-h]	block files
/dev/rhp[0-7][a-h]	raw files

SEE ALSO

hk(4), up(4)

DIAGNOSTICS

hp%d%c: hard error sn%d mbsr=%b er1=%b er2=%b. An unrecoverable error occured during transfer of the specified sector of the specified disk partition. The MASSBUS status register is printed in hexadecimal and with the error bits decoded if any error bits other than MBEXC and DTABT are set. In any case the contents of the two error registers are also printed in octal and symbolically with bits decoded. (Note that er2 is what old rp06 manuals would call er3; the terminology is that of the rm disks). The error was either unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover the error.

hp%d: write locked. The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

hp%d: not ready. The drive was spun down or off line when it was accessed. The i/o operation is not recoverable.

hp%d%c: soft ecc sn%d. A recoverable ECC error occurred on the specified sector of the specified disk partition. This happens normally a few times a week. If it happens more frequently than this the sectors where the errors are occuring should be checked to see if certain cylinders on the pack, spots on the carriage of the drive or heads are indicated.

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

DEC-standard error logging should be supported.

Bad block forwarding is not yet supported on RP06's.

A program to analyze the logged error information (even in its present reduced form) is needed.

The partition tables for the file systems should be read off of each pack, as they are never quite what any single installation would prefer, and this would make packs more portable.

NAME

ht − TM-03/TE-16,TU-45,TU-77 MASSBUS magtape interface

SYNOPSIS

formatter ht0 at mba? drive ?
tape tu0 at ht0 slave 0

DESCRIPTION

The tm-03/transport combination provides a standard tape drive interface as described in *mt*(4). All drives provide both 800 and 1600 bpi; the TE-16 runs at 45 ips, the TU-45 at 75 ips, while the TU-77 runs at 125 ips and autoloads tapes.

SEE ALSO

mt(1), tar(1), tp(1), mtop(4), mt(4), tm(4), ts(4)

DIAGNOSTICS

tu%d: no write ring. An attempt was made to write on the tape drive when no write ring was present; this message is written on the terminal of the user who tried to access the tape.

tu%d: not online. An attempt was made to access the tape while it was offline; this message is written on the terminal of the user who tried to access the tape.

tu%d: can't switch density in mid-tape. An attempt was made to write on a tape at a different density than is already recorded on the tape. This message is written on the terminal of the user who tried to switch the density.

tu%d: hard error bn%d mbsr=%b er=%b ds=%b. A tape error occurred at block *bn*, the ht error register and drive status register are printed in octal with the bits symbolically decoded. Any error is fatal on non-raw tape; when possible the driver will have retried the operation which failed several times before reporting the error.

BUGS

If any non-data error is encountered on non-raw tape, it refuses to do anything more until closed.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

NAME

lp — line printer

SYNOPSIS

device lp0 at uba0 csr 0177514 vector lpintr

DESCRIPTION

Lp provides the interface to any of the standard DEC line printers. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

The unit number of the printer is specified by the minor device after removing the low 3 bits, which act as per-device parameters. Currently only the lowest of the low three bits is interpreted: if it is set, the device is treated as having a 64-character set, rather than a full 96-character set. In the resulting half-ASCII mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

{	(-
}	)-
`	'-
		!-
~	^-

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. Lines longer than 132 characters are truncated (This is a parameter in the driver).

FILES

/dev/lp

SEE ALSO

lpr(1)

DIAGNOSTICS

None.

BUGS

Although the driver supports multiple printers this has never been tried. In any case user-level software support for multiple printers is not available.

NAME

mail — pseudo-device for mail notification

DESCRIPTION

The file **/dev/mail** is a multiplexor file maintained by the mail notification daemon. When a piece of mail is delivered the mail delivery process writes information on this file and the mail notification daemon **/etc/comsat** notifies a user of the arrival of the mail if the user desires to be notified.

SEE ALSO

biff(1)

FILES

/dev/mail

BUGS

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.

On PDP11's, the I/O page begins at location 0160000 of *kmem* and per-process data for the current process begins at 0140000. On VAX 11/780 the I/O space begins at physical address 20000000(16); on an 11/750 I/O space addresses are of the form fxxxxx(16); on all VAX'en per-process data for the current process is at virtual 7ffff000(16).

FILES

/dev/mem, /dev/kmem

BUGS

On PDP11's and VAX's, memory files are accessed one byte at a time, an inappropiate method for some device registers.

NAME

mt — TM-78/TU-78 MASSBUS magtape interface

SYNOPSIS

formatter mt0 at mba? drive ?
tape mu0 at mt0 slave 0

DESCRIPTION

The tm-78/transport combination provides a standard tape drive interface as described in *mtop*(4). The TU78 provides both 1600 and 6250 bpi, runs at 125 ips, and autoloads tapes.

SEE ALSO

mt(1), tar(1), tp(1), mtop(4), ht(4), tm(4), ts(4)

DIAGNOSTICS

mu%d: no write ring. An attempt was made to write on the tape drive when no write ring was present; this message is written on the terminal of the user who tried to access the tape.

mu%d: not online. An attempt was made to access the tape while it was offline; this message is written on the terminal of the user who tried to access the tape.

mu%d: can't switch density in mid-tape. An attempt was made to write on a tape at a different density than is already recorded on the tape. This message is written on the terminal of the user who tried to switch the density.

mu%d: hard error bn%d mbsr=%b er=%o ds=%b. A tape error occurred at block *bn*; the mt error register and drive status register are printed in octal with the drive status bits symbolically decoded. Any error is fatal on non-raw tape; when possible the driver will have retried the operation which failed several times before reporting the error.

mu%d: blank tape. An attempt was made to read or space a blank tape.

BUGS

If any non-data error is encountered on non-raw tape, it refuses to do anything more until closed.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

NAME

mt − UNIX magtape interface

DESCRIPTION

The files *mt0, ..., mt15* refer to the UNIX magtape drives, which may be on the MASSBUS using the TM03 or TM78 formatters *ht*(4) or *mt*(4), or on the UNIBUS using either the TM11, TS11, or TU45 formatters *tm*(4), *ts*(4) or *ut*(4). The following description applies to any of the transport/controller pairs. The files *mt0, ..., mt7* are low density, and *mt8, ..., mt15* are high density. On the TM03 and TM11 low density is 800bpi and high density is 1600bpi. On the TM78 low density is 1600bpi and high density is 6250bpi. On the TS11 both low and high density are 1600bpi. The files *mt0, ..., mt3* and *mt8, ..., mt11* are rewound when closed; the others are not. When a file open for writing is closed, two end-of-files are written. If the tape is not to be rewound it is positioned with the head between the two tapemarks.

A standard tape consists of a series of 1024 byte records terminated by an end-of-file. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is appropriate. The associated files are named *rmt0, ..., rmt15,* but the same minor-device considerations as for the regular files still apply. A number of other ioctl operations are available on raw magnetic tape. The following definitions are from <sys/mtio.h>:

```
/*
 * Structures and definitions for mag tape io control commands
 */

/* mag tape io control commands */
#define MTIOCTOP    (('m'<<8)|1)  /* do a mag tape op */
#define MTIOCGET    (('m'<<8)|2)  /* get mag tape status */

/* structure for MTIOCTOP - mag tape op command */
struct  mtop  {
        short   mt_op;          /* operations defined below */
        daddr_t mt_count;       /* how many of them */
};

/* operations */
#define MTWEOF      0           /* write an end-of-file record */
#define MTFSF       1           /* forward space file */
#define MTBSF       2           /* backward space file */
#define MTFSR       3           /* forward space record */
#define MTBSR       4           /* backward space record */
#define MTREW       5           /* rewind */
#define MTOFFL      6           /* rewind and put the drive offline */
#define MTNOP       7           /* no operation, sets status only */

/* structure for MTIOCGET - mag tape get status command */

struct  mtget  {
        short   mt_type;                /* type of magtape device */
/* the following two registers are grossly device dependent */
```

```
        short   mt_dsreg;               /* "drive status" register */
        short   mt_erreg;               /* "error" register */
/* end device-dependent registers */
        short   mt_resid;               /* residual count */
/* the following two are not yet implemented */
        daddr_t mt_fileno;      /* file number of current position */
        daddr_t mt_blkno;       /* block number of current position */
/* end not yet implemented */
};

/*
 * Constants for mt_type byte
 */
#define MT_ISTS         01
#define MT_ISHT         02
#define MT_ISTM         03
#define MT_ISMT         04
#define MT_ISUT         05
```

Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O seeks are ignored. A zero byte count is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?, /dev/rmt?

SEE ALSO

mt(1), tar(1), tp(1), ht(4), mt(4), tm(4), ts(4)

BUGS

NAME

newtty − summary of the "new" tty driver

USAGE

stty new

stty new crt

DESCRIPTION

This is a summary of the new tty driver, described completely, with the old terminal driver, in *tty*(4). The new driver is largely compatible with the old but provides additional functionality for job control.

CRTs and printing terminals.

The new terminal driver acts differently on CRTs and on printing terminals. On CRTs at speeds of 1200 baud or greater it normally erases input characters physically with backspace-space-backspace when they are erased logically; at speed under 1200 baud this is often unreasonably slow, so the cursor is normally merely moved to the left. This is the behavior when you say "stty new crt"; to have the tty driver always erase the characters say "stty new crt crterase crtkill", to have the characters remain even at 1200 baud or greater say "stty new crt −crterase −crtkill".

On printing terminals the command "stty new prterase" should be given. Logically erased characters are then echoed printed backwards between a '\' and an '/' character.

Other terminal modes are possible, but less commonly used; see *tty*(4) and *stty*(1) for details.

Input editing and output control.

When preparing input the character # (normally changed to ^H using *stty*(1)) erases the last input character, ^W the last input word, and the character @ (often changed to ^U) erases the entire current input line. A ^R character causes the pending input to be retyped. Lines are terminated by a return or a newline; a ^D at the beginning of a line generates an end-of-file.

Control characters echo as ^x when typed, for some x; the delete character is represented as ^?.

The character ^V may be typed before *any* character so that it may be entered without its special effect. For backwards compatibility with the old tty driver the character '\' prevents the special meaning of the character and line erase characters, much as ^V does.

Output is suspended when a ^S character is typed and resumed when a ^Q character is type. Output is discarded after a ^O character is typed until another ^O is type, more input arrives, or the condition is cleared by a program (such as the shell just before it prints a prompt.)

Signals.

A non-interactive program is interrupted by a ^? (delete); this character is often reset to ^C using *stty*(1). A quit ^\ character causes programs to terminate like ^? does, but also causes a *core* image file to be created which can then be examined with a debugger. This is often used to stop runaway processes. Interactive programs often catch interrupts and return to their command loop; only the most well debugged programs catch quits.

Programs may be stopped by hitting ^Z, which returns control to the shell. They may then be resumed using the job control mechanisms of the shell, i.e. the *fg* (foreground) command. The character ^Y is like ^Z but takes effect when read rather then when typed; it is much less frequently used.

See *tty*(4) for a more complete description of the new terminal driver.

SEE ALSO

csh(1), newcsh(1), stty(1), tty(4)

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

ra – UDA50/RA80 Unibus disk controller and drives

SYNOPSIS

controller uda0 at uba? csr 0177550 vector udintr
disk ra0 at uda0 drive 0

DESCRIPTION

The UDA50 is an intelligent disk controller for the UNIBUS. It supports the RA80 124MB disk drive.

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, etc. The standard device names begin with "ra" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The origin and size of the pseudo-disks on each drive are as follows:

RA80 124M drive partitions:

disk	start	length
ra?a	0	15884
ra?b	15884	33440
ra?c	0	237298
ra?d	0	0
ra?e	0	0
ra?f	0	0
ra?g	49324	82080
ra?h	131404	105994

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter. The ra?a partition is normally used for the root file system, the ra?b partition as a paging area, and the ra?c partition for pack-pack copying (it maps the entire disk). Both ra?g and ra?h are used to map the remaining cylinders.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/ra[0-7][a-h]	block files
/dev/rra[0-7][a-h]	raw files

SEE ALSO

hk(4), hp(4), up(4)

DIAGNOSTICS

ra%d%c: hard error sn%d udasa %o, state %d.
ra%d%c: hard error sn%d status %o. An unrecoverable error occured during transfer of the specified sector of the specified disk partition. The contents of the udasa register are printed in octal and symbolically with bits decoded. The internal state is also printed. In the second form, the MSCP status is printed. The error was either unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover the error.

ra%d%c: hard error sn%d OFFLINE. The drive was spun down or off line when it was accessed. The i/o operation is not recoverable.

uda%d: random interrupt ignored. The UDA received an interrupt before it was initialized.

uda%d: fatal error (%o). A fatal error was noticed in the UDA interrupt routine. The contents of udasa are printed. The controller will be reinitialized.

uda%d: [soft|hard] error, controller error, event 0%o.
uda%d: [soft|hard] error, host memory access error, event 0%o, addr 0%o.
uda%d: [soft|hard] error, disk transfer error, unit %d, grp %d, cyl %d, sec %d, trk %d, lbn %d, retry %d, level %d.
uda%d: [soft|hard] error, SDI error, unit %d, event 0%o, cyl %d.
uda%d: [soft|hard] error, small disk error, unit %d, event 0%o, cyl %d.
uda%d: [soft|hard] error, unknown error, unit %d, format 0%o, event 0%o.
See the driver and the UDA and MSCP manuals.

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

A program to analyze the logged error information (even in its present reduced form) is needed. Error messages should be more consistent.

The partition tables for the file systems should be read off of each pack, as they are never quite what any single installation would prefer, and this would make packs more portable.

Dumps to the swap area on system crashes are not implemented.

Booting via block zero is not (and will not be) supported. The "boot" program must be loaded from the console storage device.

Although the device supports unit numbers up to 255, the driver only supports unit numbers up to 7.

NAME

rv — Racal/Vadic ACU interface

DESCRIPTION

The racal/vadic ACU interface is provided by the files */dev/cua[01]* which is a multiplexed file, and by the daemon *dnd* which monitors the file, simulating a standard DN dialer. To place an outgoing call one forks a sub-process trying to open */dev/cul?* and then opens the corresponding file */dev/cua?* file and writes a number on it. The daemon translates the call to proper format for the Racal/Vadic interface, and monitors the progress of the call recording accounting information for later use.

The codes for the phone numbers are the same as in the DN interface:

0-9	dial 0-9
:	dial *
;	dial #
—	delay for second dial tone
<	end-of-number

The entire telephone number must be presented in a single *write* system call.

It is require that an end-of-number code be given.

FILES

/dev/cua0	virtual dialer for 300 baud dialout
/dev/cua1	virtual dialer for 1200 baud dialout
/dev/cul0	the terminal which is connected to the 300 baud dialout
/dev/cul1	the terminal which is connected to the 1200 baud dialout
/usr/adm/dnacct	Accounting records for sucessfully completed calls.

SEE ALSO

cu(1), uucp(1)

BUGS

Locking problems.

The multiplexor seems to have rare-case bugs which occasinally crash the system taking trap type 9's, usually in the *sdata* system routine.

NAME

tm — TM-11/TE-10 magtape interface

SYNOPSIS

controller tm0 at uba? csr 0172520 vector tmintr
tape te0 at tm0 drive 0

DESCRIPTION

The tm-11/te-10 combination provides a standard tape drive interface as described in *mt*(4). Hardware implementing this on the VAX is typified by the Emulex TC-11 controller operating with a Kennedy model 9300 tape transport, providing 800 and 1600 bpi operation at 125 ips.

SEE ALSO

mt(1), tar(1), tp(1), ht(4), mt(4), ts(4)

DIAGNOSTICS

te%d: no write ring. An attempt was made to write on the tape drive when no write ring was present; this message is written on the terminal of the user who tried to access the tape.

te%d: not online. An attempt was made to access the tape while it was offline; this message is written on the terminal of the user who tried to access the tape.

te%d: can't switch density in mid-tape. An attempt was made to write on a tape at a different density than is already recorded on the tape. This message is written on the terminal of the user who tried to switch the density.

te%d: hard error bn%d er=%b. A tape error occurred at block *bn*; the tm error register is printed in octal with the bits symbolically decoded. Any error is fatal on non-raw tape; when possible the driver will have retried the operation which failed several times before reporting the error.

te%d: lost interrupt. A tape operation did not complete within a reasonable time, most likely because the tape was taken off-line during rewind or lost vacuum. The controller should, but does not, give an interrupt in these cases. The device will be made available again after this message, but any current open reference to the device will return an error as the operation in progress aborts.

BUGS

If any non-data error is encountered on non-raw tape, it refuses to do anything more until closed.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

NAME

ts − TS-11 magtape interface

SYNOPSIS

controller zs0 at uba? csr 0172520 vector tsintr
tape ts0 at zs0 drive 0

DESCRIPTION

The ts-11 combination provides a standard tape drive interface as described in *mt*(4). The ts-11 operates only at 1600 bpi, and only one transport is possible per controller.

SEE ALSO

mt(1), tar(1), tp(1), ht(4), mt(4), tm(4)

DIAGNOSTICS

ts%d: no write ring. An attempt was made to write on the tape drive when no write ring was present; this message is written on the terminal of the user who tried to access the tape.

ts%d: not online. An attempt was made to access the tape while it was offline; this message is written on the terminal of the user who tried to access the tape.

ts%d: hard error bn%d xs0=%b. A hard error occurred on the tape at block *bn*, status register 0 is printed in octal and symbolically decoded as bits.

BUGS

If any non-data error is encountered on non-raw tape, it refuses to do anything more until closed.

The device lives at the same address as a tm-11 *tm*(4); as it is very difficult to get this device to interrupt, a generic system assumes that a ts is present whenever no tm-11 exists but the csr responds and a ts-11 is configured. This does no harm as long as a non-existant ts-11 is not accessed.

The system should remember which controlling terminal has the tape drive open and write error messages to that terminal rather than on the console.

NAME

tty – general terminal interface

DESCRIPTION

This section describes both a particular special file **/dev/tty** and the terminal drivers used for conversational computing.

Line disciplines.

The system provides different *line disciplines* for controlling communications lines. In this version of the system there are three disciplines available:

old — The old (standard) terminal driver. This is used when using the standard shell *sh*(1) and for compatibility with other standard version 7 UNIX systems.

new — A newer terminal driver, with features for job control; this must be used when using *csh*(1). See *newtty*(1) for a short user-level summary.

net — A line discipline used for networking and loading data into the system over communications lines. It allows high speed input at very low overhead, and is described in *bk*(4).

Line discipline switching is accomplished with the TIOCSETD *ioctl:*

int ldisc = LDISC; ioctl(filedes, TIOCSETD, &ldisc);

where LDISC is OTTYDISC for the standard tty driver, NTTYDISC for the new driver and NETLDISC for the networking discipline. The standard (currently old) tty driver is discipline 0 by convention. The current line discipline can be obtained with the TIOCGETD ioctl. Pending input is discarded when the line discipline is changed.

All of the low-speed asynchronous communications ports can use any of the available line disciplines, no matter what hardware is involved. The remainder of this section discusses the "old" and "new" disciplines.

The control terminal.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by *init*(8) and become a user's standard input and output file.

If a process which has no control terminal opens a terminal file, then that terminal file becomes the control terminal for that process. The control terminal is thereafter inherited by a child process during a *fork*(2), even if the control terminal is closed.

The file **/dev/tty** is, in each process, a synonym for a *control terminal* associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

Process groups.

As described more completely in *jobs*(3), command processors such as *csh*(1) can arbitrate the terminal between different *jobs* by placing related jobs in a single process group and associating this process group with the terminal. A terminals associated process group may be set using the TIOCSPGRP *ioctl*(2):

ioctl(fildes, TIOCSPGRP, &pgrp)

or examined using TIOCGPGRP rather than TIOCSPGRP, returning the current process group in *pgrp.* The new terminal driver aids in this arbitration by restricting access to the terminal by processes which are not in the current process group; see **Job access control** below.

Modes.

The terminal drivers have three major modes, characterized by the amount of processing on the input and output characters:

cooked The normal mode. In this mode lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when an EOT (control-D, hereafter ^D) is entered. A carriage return is usually made synonymous with newline in this mode, and replaced with a newline whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, etc.) are available in this mode.

CBREAK This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next and interrupt processing are still done in this mode. Output processing is done.

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be very different when, in the new terminal driver, a process asks for notification via a SIGTTIN *signal*(2) when input is ready to be read from the control terminal. In this case a *read*(2) from the control terminal will never block, but rather return an error indication (EIO) if there is no input available.

Input editing.

A UNIX terminal 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. In the old terminal driver all the saved characters are thrown away when the limit is reached, without notice; the new driver simply refuses to accept any further input, and rings the terminal bell.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word it is possible to have input characters with that parity discarded (see the **Summary** below.)

In all of the line disciplines, it is possible to simulate terminal input using the TIOCSTI ioctl, which takes, as its third argument, the address of a character. The system pretends that this character was typed on the argument terminal, which must be the control terminal except for the super-user (this call is not in standard version 7 UNIX)..

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the *stty*(2) call or the TIOCSETN or TIOCSETP ioctls (see the **Summary** below).

In cooked mode, terminal input is processed in units of lines. A program attempting to read will normally be suspended until an entire line has been received (but see the description of SIGTTIN in **Modes** above and FIONREAD in **Summary** below.) No matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, line editing is normally done, with the character '#' logically erasing the last character typed and the character '@' logically erasing the entire current input line. These are often reset on crt's, with ^H replacing #, and ^U replacing @. These characters never erase beyond the beginning of the current input line or an ^D. These characters may be entered literally by preceding them with '\'; in the old teletype driver both the '\' and the character entered literally will appear on the screen; in the new driver the '\' will normally disappear.

The drivers normally treat either a carriage return or a newline character as terminating an input line, replacing the return with a newline and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word then the processing for carriage return is disabled, and it is simply echoed as a return, and does not terminate cooked mode input.

In the new driver there is a literal-next character ^V which can be typed in both cooked and CBREAK mode preceding **any** character to prevent its special meaning. This is to be preferred to the use of '\' escaping erase and kill characters, but '\' is (at least temporarily) retained with its old function in the new driver for historical reasons.

The new terminal driver also provides two other editing characters in normal mode. The word-erase character, normally ^W, erases the preceding word, but not any spaces before it. For the purposes of ^W, a word is defined as a sequence of non-blank characters, with tabs counted as blanks. Finally, the reprint character, normally ^R, retypes the pending input beginning on a new line. Retyping occurs automagically in cooked mode if characters which would normally be erased from the screen are fouled by program output.

Input echoing and redisplay

In the old terminal driver, nothing special occurs when an erase character is typed; the erase character is simply echoed. When a kill character is typed it is echoed followed by a new-line (even if the character is not killing the line, because it was preceded by a '\'!.)

The new terminal driver has several modes for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy terminals. When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters which are logically erased are then printed out backwards preceded by '\' and followed by '/' in this mode.

Crt terminals. When a crt terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal driver then echoes the proper number of erase characters when input is erased; in the normal case where the erase character is a ^H this causes the cursor of the terminal to back up to where it was before the logically erased character was typed. If the input has become fouled due to interspersed asynchronous output, the input is automagically retyped.

Erasing characters from a crt. When a crt terminal is in use, the LCRTERA bit may be set to cause input to be erased from the screen with a "backspace-space-backspace" sequence when character or word deleting sequences are used. A LCRTKIL bit may be set as well, causing the input to be erased in this manner on line kill sequences as well.

Echoing of control characters. If the LCTLECH bit is set in the local state word, then non-printing (control) characters are normally echoed as ^X (for some X) rather than being echoed unmodified; delete is echoed as ^?.

The normal modes for using the new terminal driver on crt terminals are speed dependent. At speeds less than 1200 baud, the LCRTERA and LCRTKILL processing is painfully slow, so *stty*(1) normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater all of these bits are normally set. *Stty*(1) summarizes these option settings and the use of the new terminal driver as "newcrt."

Output processing.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. (As noted above, input characters are normally 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 normally generated on output. The EOT character is not transmitted in cooked mode to prevent terminals that respond to it from hanging up; programs using raw or cbreak mode should be careful.

The terminal drivers provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces ^H, form feeds ^L, carriage returns ^M, tabs ^I and newlines ^J. The driver will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns. These functions are controlled by bits in the tty flags word; see **Summary** below.

The terminal drivers provide for mapping between upper and lower case on terminals lacking lower case, and for other special processing on deficient terminals.

Finally, in the new terminal driver, there is a output flush character, normally ^O, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. Ioctls to flush the characters in the input and output queues TIOCFLUSH, and to return the number of character still in the output queue TIOCOUTQ are also available.

Upper case terminals and Hazeltines

If the LCASE bit is set in the tty flags, then all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by '\'. If the new terminal driver is being used, then upper case letters are preceded by a '\' when output. In addition, the following escape sequences can be generated on output and accepted on input:

for	`	|	~	{	}
use	\'	\!	\^	\(	\)

To deal with Hazeltine terminals, which do not understand that ~ has been made into an ASCII character, the LTILDE bit may be set in the local mode word when using the new terminal driver; in this case the character ~ will be replaced with the character ` on output.

Flow control.

There are two characters (the stop character, normally ^S, and the start character, normally ^Q) which cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode the system produces a stop character (default ^S) when the input queue is in danger of overflowing, and a start character (default ^Q) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys the conventions.

Line control and breaks.

There are several *ioctl* calls available to control the state of the terminal line. The TIOCSBRK ioctl will set the break bit in the hardware interface causing a break condition to exist; this can be cleared (usually after a delay with *sleep*(3)) by TIOCCBRK. Break conditions in the input are reflected as a null character in RAW mode or as the interrupt character in cooked or CBREAK mode. The TIOCCDTR ioctl will clear the data terminal ready condition; it can be set again by TIOCSDTR.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a SIGHUP hangup signal is sent to the processes in the distinguished process group of the terminal; this usually causes them to terminate (the SIGHUP can be suppressed by setting the LNOHANG bit in the local state word of the driver.) Access to the terminal by other processes is then normally revoked, so any further reads will fail, and programs that read a terminal and test for end-of-file on their input will terminate appropriately.

When using an ACU it is possible to ask that the phone line be hung up on the last close with the TIOCHPCL ioctl; this is normally done on the outgoing line.

Interrupt characters.

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent the processes in the control group of the terminal, as if a TIOCGPGRP ioctl were done to get the process group and then a *killpg*(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (*à 'la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed, although this is not often done.

^? **t_intrc** (Delete) generates a SIGINTR signal. This is the normal way to stop a process which is no longer interesting, or to regain control in an interactive program.

^\ **t_quitc** (FS) generates a SIGQUIT signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.

^Z **t_suspc** (EM) generates a SIGTSTP signal, which is used to suspend the current process group.

^Y **t_dstopc** (SUB) generates a SIGTSTP signal as ^Z does, but the signal is sent when a program attempts to read the ^Y, rather than when it is typed.

Job access control.

When using the new terminal driver, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal its process group is sent a SIGTTIN signal, which normally causes the members of that process group to stop. If, however, the process is ignoring or holding SIGTTIN signal is an orphan its parent has exited and it has been inherited by the *init*(8) process, or if it is a process in the middle of process creation using *vfork*(2)), it is instead returned an end-of-file. Under older UNIX systems these processes would typically have had their input files reset to **/dev/null**, so this is a compatible change.

When using the new terminal driver with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal if it is not in the distinguished process group for that terminal. Processes which are holding or ignoring SIGTTOU signals, which are orphans, or which are in the middle of a *vfork*(2) are excepted and allowed to produce output.

Summary of modes.

Unfortunately, due to the evolution of the terminal driver, there are 4 different structures which contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between version 6 and version 7 UNIX systems. The second contains additional control characters added in version 7. The third is a word of local state peculiar to the new terminal driver, and the fourth is another structure of special characters added for the new driver. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty.

The basic *ioctls* use the structure defined in <*sgtty.h*>:

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
```

```
        char    sg_kill;
        short   sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <*sgtty.h*>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are # 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 suitable for the concept-100 and pads lines to be at least 9 characters at 9600 baud.

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.

Input characters with the wrong parity, as determined by bits 200 and 100, are ignored in cooked and CBREAK mode.

RAW disables all processing save output flushing with LFLUSHO; full 8 bits of input are given as soon as it is available; all 8 bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode it is discarded; this applies to both new and old drivers.

CRMOD causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a new-line function).

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ or EOT are disabled.

TANDEM mode causes the system to produce a stop character (default ^S) whenever the input queue is in danger of overflowing, and a start character (default ^Q) when the input queue has drained sufficiently. It is useful for flow control when the 'terminal' is really another computer which understands the conventions.

Basic ioctls

In addition to the TIOCSETD and TIOCGETD disciplines discussed in **Line disciplines** above, a large number of other *ioctl*(2) calls apply to terminals, and have the general form:

```
#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;
```

The applicable codes are:

TIOCGETP Fetch the basic parameters associated with the terminal, and store in the pointed-to *sgttyb* structure.

TIOCSETP Set the parameters according to the pointed-to *sgttyb* structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN Set the parameters like TIOCSETP but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes the *arg* is ignored.

TIOCEXCL Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL Turn off "exclusive-use" mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

TIOCFLUSH All characters waiting in input or output queues are flushed.

The remaining calls are not available in vanilla version 7 UNIX. In cases where arguments are required, they are described; *arg* should otherwise be given as 0.

TIOCSTI the argument is the address of a character which the system pretends was typed on the terminal.

TIOCSBRK the break bit is set in the terminal.

TIOCCBRK the break bit is cleared.

TIOCSDTR data terminal ready is set.

TIOCCDTR data terminal ready is cleared.

TIOCGPGRP arg is the address of a word into which is placed the process group number of the control terminal.

TIOCSPGRP arg is a word (typically a process id) which becomes the process group for the control terminal.

FIONREAD returns in the long integer whose address is arg the number of immediately readable characters from the argument unit. This works for files, pipes, and terminals, but not (yet) for multiplexed channels.

Tchars

The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces: The following structure is defined in <*sys/ioctl.h*>, which is automatically included 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 ^?, ^\, ^Q, ^S, ^D, 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 applicable ioctl calls are:

TIOCGETC Get the special characters and put them in the specified structure.

TIOCSETC Set the special characters to those given in the structure.

Local mode

The third structure associated with each terminal is a local mode word; except for the LNOHANG bit, this word is interpreted only when the new driver is in use. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as backspace-space-backspace
LTILDE	000010	Convert ~ to ` on output (for Hazeltine terminals)
LMDMBUF	000020	Stop/start output when carrier drops
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
LETXACK	001000	Diablo style buffer hacking (unimplemented)
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LINTRUP	004000	Generate interrupt SIGTINT when input ready to read
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only ^Q restarts output after ^S, like DEC systems

The applicable *ioctl* functions are:

TIOCLBIS arg is the address of a mask which is the bits to be set in the local mode word.

TIOCLBIC arg is the address of a mask of bits to be cleared in the local mode word.

TIOCLSET arg is the address of a mask to be placed in the local mode word.

TIOCLGET arg is the address of a word into which the current mask is placed.

Local special chars

The final structure associated with each terminal is the *ltchars* structure which defines interrupt characters for the new terminal driver. Its structure is:

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dstopc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
        char    t_werasec;      /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

The default values for these characters are ^Z, ^Y, ^R, ^O, ^W, and ^V. A value of −1 disables the character.

The applicable *ioctl* functions are:

TIOCSLTC args is the address of a *ltchars* structure which defines the new local special characters.

TIOCGLTC args is the address of a *ltchars* structure into which is placed the current set of local special characters.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

csh(1), stty(1), ioctl(2), signal(2), sigsys(2), stty(2), getty(8), init(8), newtty(4)

BUGS

Half-duplex terminals are not supported.

NAME

tu − console TU58 interface

DESCRIPTION

This is a simple interface to the TU58 cassette tape unit, which is part of the console subsytem for VAX-11/750's. Access is given to the entire tape consisting of 1024 blocks of 512 bytes.

The TU58 behaves like a disk but has only a block interface (no raw). The cassette is commonly accessed via the *arff*(8) command, using the 'm' flag.

FILES

/dev/floppy

SEE ALSO

arff(8)

DIAGNOSTICS

Lots, mostly cryptic. Read the driver.

BUGS

The driver is very unreliable due to the nature of the TU58 interface. The TU58 should be used only when the system is in single-user mode, since it is often necessary to reboot the system to unhang the process accessing the TU58.

NAME

up — unibus storage module controller/drives

SYNOPSIS

controller sc0 at uba? csr 0176700 vector upintr
disk up0 at sc0 drive 0

DESCRIPTION

This is a generic UNIBUS storage module disk driver. It is specifically designed to work with the Emulex SC-21 controller and Ampex or CDC 300M or Fujitsu 160M drives. It can be easily adapted to other drives and controllers (although bootstrapping will not necessarily be directly possible.)

Files with minor device numbers 0 through 7 refer to various portions of drive 0; minor devices 8 through 15 refer to drive 1, etc. The standard device names begin with "up" followed by the drive number and then a letter a-h for partitions 0-7 respectively. The character ? stands here for a drive number in the range 0-7.

The origin and size of the pseudo-disks on each drive are as follows:

AMPEX/CDC 300M drive partitions:

disk	start	length	cyl
up?a	0	15884	0-26
up?b	16416	33440	27-81
up?c	0	495520	0-814
up?d	341696	15884	562-588
up?e	358112	55936	589-680
up?f	414048	81472	681-814
up?g	341696	153824	562-814
up?h	49856	291346	82-561

FUJITSU 160M drive partitions:

disk	start	length	cyl
up?a	0	15884	0-49
up?b	16000	33440	50-154
up?c	0	263360	0-822
up?h	49600	213760	155-822

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter. The up?a partition is normally used for the root file system, the up?b partition as a paging area, and the up?c partition for pack-pack copying (it maps the entire disk). On 160M drives the up?h partition maps the rest of the pack. On 300M drives both up?g and up?h are used to map the remaining cylinders.

The block files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a 'raw' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw files conventionally begin with an extra 'r.'

In raw I/O counts should be a multiple of 512 bytes (a disk sector). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/up[0-7][a-h]	block files
/dev/rup[0-7][a-h]	raw files

SEE ALSO

hk(4), hp(4)

DIAGNOSTICS

up%d%c: hard error sn%d cs2=%b er1=%b er2=%b. An unrecoverable error occured during transfer of the specified sector of the specified disk partition. The contents of the cs2, er1 and er2 registers are printed in octal and symbolically with bits decoded. The error was either unrecoverable, or a large number of retry attempts (including offset positioning and drive recalibration) could not recover the error.

up%d: write locked. The write protect switch was set on the drive when a write was attempted. The write operation is not recoverable.

up%d: not ready. The drive was spun down or off line when it was accessed. The i/o operation is not recoverable.

up%d: not ready (came back!). The drive was not ready, but after printing the message about being not ready (which takes a fraction of a second) was ready. The operation is recovered if no further errors occur.

up%d%c: soft ecc sn%d. A recoverable ECC error occurred on the specified sector of the specified disk partition. This happens normally a few times a week. If it happens more frequently than this the sectors where the errors are occuring should be checked to see if certain cylinders on the pack, spots on the carriage of the drive or heads are indicated.

sc%d: lost interrupt. A timer watching the controller detecting no interrupt for an extended period while an operation was outstanding. This indicates a hardware or software failure. There is currently a hardware/software problem with spinning down drives while they are being accessed which causes this error to occur. The error causes a UNIBUS reset, and retry of the pending operations. If the controller continues to lose interrupts, this error will recur a few seconds later.

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 512-byte block boundaries, and *write* scribbles on the tail of incomplete blocks. Thus, in programs that are likely to access raw devices, *read, write* and *lseek*(2) should always deal in 512-byte multiples.

DEC-standard error logging and bad block forwarding should be supported; the code to do this could be easily incorporated from the *hp*(4) driver. All that would be needed then would be a stand-alone formatting program to detect the bad sectors, format the disk so that the sectors were marked bad and initialize the bad sector files.

A program to analyze the logged error information (even in its present reduced form) is needed.

The partition tables for the file systems should be read off of each pack, as they are never quite what any single installation would prefer, and this would make packs more portable.

NAME

va – Benson-Varian interface

SYNOPSIS

device va0 at uba? csr 0164000 vector vaintr

DESCRIPTION

The Benson-Varian printer/plotter in normally used with the programs *vpr*(1), *vprint*(1) or *vtroff*(1). This description is designed for those who wish to drive the Benson-Varian directly.

The Benson-Varian at Berkeley uses 11" by 8" fan-fold paper. It will print 132 characters per line in print mode and 2112 dots per line in plot mode.

In print mode, the Benson-Varian uses a modified ASCII character set. Most control characters print various non-ASCII graphics such as daggers, sigmas, copyright symbols, etc. Only LF and FF are used as format effectors. LF acts as a newline, advancing to the beginning of the next line, and FF advances to the top of the next page.

In plot mode, the Benson-Varian prints one raster line at a time. An entire raster line of bits (2112 bits = 264 bytes) is sent, and then the Benson-Varian advances to the next raster line.

Note: The Benson-Varian must be sent an even number of bytes. If an odd number is sent, the last byte will be lost. Nulls can be used in print mode to pad to an even number of bytes.

To use the Benson-Varian yourself, you must realize that you cannot open the device, */dev/va0* if there is a daemon active. You can see if there is a daemon active by doing a *ps*(1), or by looking in the directory */usr/spool/vad.* If there is a file *lock* there, then there is probably a daemon */usr/lib/vad* running. If not, you should remove the *lock.*

In any case, when your program tries to open the device */dev/va0* you may get one of two errors. The first of these ENXIO indicates that the Benson-Varian is already in use. Your program can then *sleep*(2) and try again in a while, or give up. The second is EIO and indicates that the Benson-Varian is offline.

To set the Benson-Varian into plot mode, "#include <vcmd.h>" and use the following *ioctl*(2) call

```
ioctl(fileno(va), VSETSTATE, plotmd);
```

where **plotmd** is defined to be

```
int plotmd[] = { VPLOT, 0, 0 };
```

and *va* is the result of a call to *fopen* on stdio. When you finish using the Benson-Varian in plot mode you should advance to a new page by sending it a FF after putting it back into print mode, i.e. by

```
int prtmd[] = { VPRINT, 0, 0 };
...
fflush(va);
ioctl(fileno(va), VSETSTATE, prtmd);
write(fileno(va), "\f\0", 2);
```

N.B.: If you use the standard I/O library with the Benson-Varian you **must** do

```
setbuf(vp, vpbuf);
```

where *vpbuf* is declared

```
char vpbuf[BUFSIZ];
```

otherwise the standard I/O library, thinking that the Benson-Varian is a terminal (since it is a character special file) will not adequately buffer the data you are sending to the Benson-Varian. This will cause it to run extremely slowly and tend to grind the system to a halt.

FILES

/dev/va0
/usr/include/sys/vcmd.h

SEE ALSO

vfont(5), vpr(1), vtroff(1), vp(4)

DIAGNOSTICS

va%d: npr timeout. The device was not able to get data from the UNIBUS within the timeout period, most likely because some other device was hogging the bus. (But see BUGS below).

BUGS

The 1's (one's) and l's (lower-case el's) in the Benson-Varian's standard character set look very similar; caution is advised.

The interface hardware is rumored to have problems which can play havoc with the UNIBUS. We have intermittent minor problems on the UNIBUS where our va lives, but haven't ever been able to pin them down completely.

NAME

vp — Versatec interface

SYNOPSIS

device vp0 at uba? csr 0177510 vector vpintr vpintr

DESCRIPTION

The Versatec printer/plotter is normally used with the programs *vpr*(1), *vprint*(1) or *vtroff*(1). This description is designed for those who wish to drive the Versatec directly.

The Versatec at Berkeley is 36" wide, and has 440 characters per line and 7040 dots per line in plot mode (this is actually slightly less than 36" of dots.) The paper used is continuous roll paper, and comes in 500' rolls.

To use the Versatec yourself, you must realize that you cannot open the device, */dev/vp0* if there is a daemon active. You can see if there is a daemon active by doing a *ps*(1), or by looking in the directory */usr/spool/vpd.* If there is a file *lock* there, then there is probably a daemon */usr/lib/vpd* running. If not, you should remove the *lock.*

In any case, when your program tries to open the device */dev/vp0* you may get one of two errors. The first of these ENXIO indicates that the Versatec is already in use. Your program can then *sleep*(2) and try again in a while, or give up. The second is EIO and indicates that the Versatec is offline.

To set the Versatec into plot mode you should include <vcmd.h> and use the *ioctl*(2) call

```
ioctl(fileno(vp), VSETSTATE, plotmd);
```

where **plotmd** is defined to be

```
int plotmd[] = { VPLOT, 0, 0 };
```

and *vp* is the result of a call to *fopen* on stdio. When you finish using the Versatec in plot mode you should eject paper by sending it a EOT after putting it back into print mode, i.e. by

```
int prtmd[] = { VPRINT, 0, 0 };
...
fflush(vp);
ioctl(fileno(vp), VSETSTATE, prtmd);
write(fileno(vp), "\04", 1);
```

N.B.: If you use the standard I/O library with the Versatec you **must** do

```
setbuf(vp, vpbuf);
```

where *vpbuf* is declared

```
char vpbuf[BUFSIZ];
```

otherwise the standard I/O library, thinking that the Versatec is a terminal (since it is a character special file) will not adequately buffer the data you are sending to the Versatec. This will cause it to run extremely slowly and tends to grind the system to a halt.

FILES

/dev/vp0

SEE ALSO

vfont(5), vpr(1), vtroff(1), va(4)

DIAGNOSTICS

None.

BUGS

The configuration part of the driver assumes that the device is setup to vector print mode through 0174 and plot mode through 0200. Since the driver doesn't care whether the device considers the interrupt to be a print or a plot interrupt, it would be preferable to have these be the same. This since the configuration program can't be sure at boot time which vector interrupted and where the interrupt vectors actually are. For the time being, since our versatec is vectored as described above, we specify that it has two interrupt vectors and are careful to detect an interrupt through 0200 at boot time and (manually) pretend the interrupt came through 0174.

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 editor *ld*(1). Both programs make *a.out* executable if there were no errors and no unresolved external references. Layout information as given in the include file for the VAX-11 is:

```
/*
 * Header prepended to each a.out file.
 */
struct exec {
        long     a_magic;  /* magic number */
        unsigned a_text;   /* size of text segment */
        unsigned a_data;   /* size of initialized data */
        unsigned a_bss;    /* size of uninitialized data */
        unsigned a_syms;   /* size of symbol table */
        unsigned a_entry;  /* entry point */
        unsigned a_trsize; /* size of text relocation */
        unsigned a_drsize; /* size of data relocation */
};

#define  OMAGIC 0407       /* old impure format */
#define  NMAGIC 0410       /* read-only text */
#define  ZMAGIC 0413       /* demand load format */

/*
 * Macros which take exec structures as arguments and tell whether
 * the file has a reasonable magic number or offsets to text|symbols|strings.
 */
#define  N_BADMAG(x) \
   (((x).a_magic)!=OMAGIC && ((x).a_magic)!=NMAGIC && ((x).a_magic)!=ZMAGIC)

#define  N_TXTOFF(x) \
        ((x).a_magic==ZMAGIC ? 1024 : sizeof (struct exec))
#define N_SYMOFF(x) \
        (N_TXTOFF(x) + (x).a_text+(x).a_data + (x).a_trsize+(x).a_drsize)
#define  N_STROFF(x) \
        (N_SYMOFF(x) + (x).a_syms)
```

The file has five sections: a header, the program text and data, relocation information, a symbol table and a string table (in that order). The last three may be omitted 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. The size of the header is not included in any of the other sizes.

When an *a.out* file is executed, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number in the header is OMAGIC (0407), 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. This is the oldest kind of executable program and is rarely used. If the magic number is NMAGIC (0410) or ZMAGIC (0413), the data segment begins at the first 0 mod 1024 byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. For ZMAGIC format, the text segment begins at a 0 mod 1024 byte boundary in the *a.out* file, the remaining bytes after the header in the first block are reserved and should be zero. In this case the text and data sizes must both be multiples of 1024 bytes, and the pages of the file will be brought into the running image as needed, and not pre-loaded as with the other formats. This is especially suitable for very large programs and is the default format produced by *ld*(1).

The stack will occupy the highest possible locations in the core image: growing downwards from 0x7ffff000. The stack is automatically extended as required. The data segment is only extended as requested by *break*(2).

After the header in the file follow the text, data, text relocation data relocation, symbol table and string table in that order. The text begins at the byte 1024 in the file for ZMAGIC format or just after the header for the other formats. The N_TXTOFF macro returns this absolute file position when given the name of an exec structure as argument. The data segment is contiguous with the text and immediately followed by the text relocation and then the data relocation information. The symbol table follows all this; its position is computed by the N_SYMOFF macro. Finally, the string table immediately follows the symbol table at a position which can be gotten easily using N_STROFF. The first 4 bytes of the string table are not used for string storage, but rather contain the size of the string table; this size INCLUDES the 4 bytes, the minimum string table size is thus 4.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file as follows:

```
/*
 * Format of a symbol table entry.
 */
struct nlist {
        union {
            char        *n_name;  /* for use when in-core */
            long        n_strx;   /* index into file string table */
        } n_un;
        unsigned char n_type;     /* type flag, i.e. N_TEXT etc; see below */
        char          n_other;
        short         n_desc;     /* see <stab.h> */
        unsigned      n_value;    /* value of this symbol (or sdb offset) */
};
#define n_hash        n_desc      /* used internally by ld */

/*
 * Simple values for n_type.
 */
#define N_UNDF        0x0         /* undefined */
#define N_ABS         0x2         /* absolute */
#define N_TEXT        0x4         /* text */
#define N_DATA        0x6         /* data */
#define N_BSS         0x8         /* bss */
#define N_COMM        0x12        /* common (internal to ld) */
#define N_FN          0x1f        /* file name symbol */

#define N_EXT         01          /* external bit, or'ed in */
```

```
#define N_TYPE      0x1e      /* mask for all the type bits */

/*
 * Other permanent symbol table entries have some of the N_STAB bits set.
 * These are given in <stab.h>
 */
#define N_STAB      0xe0      /* if any of these bits set, don't discard */

/*
 * Format for namelist values.
 */
#define N_FORMAT "%08x"
```

In the *a.out* file a symbol's n_un.n_strx field gives an index into the string table. A n_strx value of 0 indicates that no name is associated with a particular symbol table entry. The field n_un.n_name can be used to refer to the symbol name only if the program sets this up using n_strx and appropriate data from the string table.

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 byte in the text or data which is not a portion of a reference to an undefined external symbol is exactly that value which will appear in memory when the file is executed. If a byte in the text or data involves a reference to an undefined external symbol, as indicated by the relocation information, then the value 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 to the bytes in the file.

If relocation information is present, it amounts to eight bytes per relocatable datum as in the following structure:

```
/*
 * Format of a relocation datum.
 */
struct relocation_info {
        int       r_address;          /* address which is relocated */
        unsigned  r_symbolnum:24,     /* local symbol ordinal */
                  r_pcrel:1,          /* was relocated pc relative already */
                  r_length:2,         /* 0=byte, 1=word, 2=long */
                  r_extern:1,         /* does not include value of sym referenced */
                  :4;                 /* nothing, yet */
};
```

There is no relocation information if a_trsize+a_drsize==0. If r_extern is 0, then r_symbolnum is actually a n_type for the relocation (i.e. N_TEXT meaning relative to segment text origin.)

SEE ALSO

adb(1), as(1), ld(1), nm(1), sdb(1), stab(5), strip(1)

BUGS

Not having the size of the string table in the header is a loss, but expanding the header size would have meant stripped executable file incompatibility, and we couldn't hack this just now.

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:

```
/*	acct.h	3.2		6/6/80*/

/*
 * 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

aliases — aliases file for delivermail

SYNOPSIS

/usr/lib/aliases

DESCRIPTION

This file describes user id aliases that will be used by */etc/delivermail.* It is formatted as a series of lines of the form

name:addr1,addr2,...addrn

The *name* is the name to alias, and the *addri* are the addresses to send the message to. Lines beginning with white space are continuation lines. Lines beginning with '#' are comments.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

This is only the raw data file; the actual aliasing information is placed into a binary format in the files /usr/lib/aliases.dir and /usr/lib/aliases.pag using the program *newaliases*(5). A *newaliases* command should be executed each time the aliases file is changed for the change to take effect.

SEE ALSO

newaliases(1), dbm(3), delivermail(8)

BUGS

Because of restrictions in *dbm*(3) a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by "chaining"; i.e. make the last name in the alias by a dummy name which is a continuation alias.

NAME

ar — archive (library) file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

N.B.: This archive format is new to this distribution. See *old*(8) and *arcv*(1) for programs to deal with the old format.

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 string at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARMAG  "!<arch>\n"
#define SARMAG 8

#define ARFMAG "`\n"

struct ar_hdr {
        char    ar_name[16];
        char    ar_date[12];
        char    ar_uid[6];
        char    ar_gid[6];
        char    ar_mode[8];
        char    ar_size[10];
        char    ar_fmag[2];
};
```

The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. 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 a 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.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

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

core − format of memory image file

DESCRIPTION

UNIX writes out a memory 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 memory image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The maximum size of a *core* file is limited by *vlimit*(2). Files which would be larger than the limit are not created.

The core file consists of the *u.* area, which currently consists of 6 pages, beginning with a *user* structure as given in /usr/include/sys/user.h. The kernel stack grows from the end of this 6 page region. The remainder of the core file consists first of the data pages and then the stack pages of the process image.

In general the debugger *adb*(1) is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2), vlimit(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 entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory of the master file system ("/"), 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;
        int          c_magic;
        int          c_checksum;
        struct       dinode          c_dinode;
        int          c_count;
        char         c_addr[BSIZE];
} spcl;

struct  idates {
        char         id_name[16];
        char         id_incno;
        time_t       id_ddate;
};

#define DUMPOUTFMT      "%-16s %c %s"          /* for printf */
                                          /* name, incno, ctime(date) */
#define DUMPINFMT  "%16s %c %[^\n]\n"      /* inverse for scanf */
```

NTREC is the number of 1024 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 (1024-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(8), dumpdir(8), restor(8), 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=:/usr/ucb:/bin:/usr/bin.

HOME A user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff* or *plot*(1), which may exploit special terminal capabilities. See */etc/termcap* (*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),*termlib*(3).

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), termlib(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. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 1024-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:

```
/*	filsys.h	3.2		6/6/80*/

/*
 * 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 */
	/* begin not maintained by this version of the system */
	short	s_dinfo[2];		/* interleave stuff */
#define		s_m			s_dinfo[0]
#define		s_n			s_dinfo[1]
	char	s_fname[6];		/* file system name */
	char	s_fpack[6];		/* file system pack name */
	/* end not maintained */
	ino_t	s_lasti;		/* start place for circular search */
	ino_t	s_nbehind;		/* est # free inodes before s_lasti */
};

#ifdef KERNEL
struct filsys *getfs();
#endif
```

S_isize is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus the i-list is *s_isize*−2 blocks long. *S_fsize* is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free[1]*, ... , *s_free[s_nfree−1]*, up to NICFREE free block numbers. NICFREE is a configuration constant. *S_free[0]* is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file <*sys/fblk.h*> is:

```
/*	fblk.h	3.2		6/6/80*/

struct fblk {
	int	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.

The fields *s_lasti* and *s_nbehind* are used to avoid searching the inode list from the beginning each time the system runs out of inodes. *S_lasti* gives the base of the block of inodes last searched on the filesystem when inodes ran out, and *s_nbehind* gives the number of inodes, whose numbers were less than *s_lasti* when they were freed with *s_ninode* already NICINODE. Thus *s_ninode* is the number of free inodes before *s_lasti*. The system will search forward for free inodes from *s_lasti* for more inodes unless *s_nbehind* is sufficiently large, in which case it will search the file system inode list from the beginning. This mechanism serves to avoid n**2 behavior in allocating inodes.

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 16 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:

```
/*	ino.h	3.2		6/6/80*/

/*
 * 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 */
};
/*
 * the 40 address bytes:
 *      39 used; 13 addresses
 *      of 3 bytes each.
 */
```

Di_mode tells the kind of file; it is encoded identically to the *st_mode field of stat*(2). *Di_nlink* is the number of directory entries (links) that refer to this i-node. *Di_uid* and *di_gid* are the owner's user and group IDs. *Size* is the number of bytes in the file. *Di_atime* and *di_mtime* are the times of last access and modification of the file contents (read, write or create) (see *times*(2)); *Di_ctime* records the time of last modification to the inode or to the file, and is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the *di_addr* field is occupied by the device code (see *types*(5)). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array *di_addr* packed into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresses are singly, doubly, and triply indirect and point to blocks of 256 block pointers. Pointers in indirect blocks have the type *daddr_t* (see *types*(5)).

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

SEE ALSO

icheck(1), dcheck(1), dir(5), mount(1), stat(2), types(5)

NAME

fstab – static information about the filesystems

SYNOPSIS

#include <fstab.h>

DESCRIPTION

The file */etc/fstab* contains descriptive information about the various file systems. */etc/fstab* is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file.

These programs use */etc/fstab: dump, mount, umount, swapon, fsck* and *df.* The order of records in */etc/fstab* is important, for both *fsck, mount,* and *umount* sequentially iterate through */etc/fstab* doing their thing.

The special file name is the **block** special file name, and not the character special file name. If a program needs the character special file name, the program must create it by appending a "r" after the last "/" in the special file name.

If *fs_type* is "rw" or "ro" then the file system whose name is given in the *fs_file* field is normally mounted read-write or read-only on the specified special file. The *fs_freq* field is used for these file systems by the *dump*(8) command to determine which file systems need to be dumped. The *fs_passno* field is used by the *fsck*(8) program to determine the order in which file system checks are done at reboot time. The root file system should be specified with a *fs_passno* of 1, and other file systems should have larger numbers. File systems within a drive should have distinct numbers, but file systems on different drives can be checked on the same pass to utilize parallelism available in the hardware.

If *fs_type* is "sw" then the special file is made avaiable as a piece of swap space by the *swapon*(8) command at the end of the system reboot procedure. The fields other than *fs_spec* and *fs_type* are not used in this case.

Fs_type may be specified as "xx" to cause an entry to be ignored. This is useful to show disk partitions which are currently not used but will be used later.

```
#define FSTAB           "/etc/fstab"
#define FSNMLG          16

#define FSTABFMT    "%16s:%16s:%2s:%d:%d\n"
#define FSTABARG(p) (p)->fs_spec, (p)->fs_file, \
                    (p)->fs_type, &(p)->fs_freq, &(p)->fs_passno
#define FSTABNARGS 5

#define FSTAB_RW    "rw"    /* read write device */
#define FSTAB_RO    "ro"    /* read only device */
#define FSTAB_SW    "sw"    /* swap device */
#define FSTAB_XX    "xx"    /* ignore totally */

struct fstab {
        char fs_spec[FSNMLG]; /* block special device name */
        char fs_file[FSNMLG]; /* file system path prefix */
        char fs_type[3];      /* rw,ro,sw or xx */
        int  fs_freq;         /* dump frequency, in days */
        int  fs_passno;       /* pass number on parallel dump */
};
```

The proper way to read records from */etc/fstab* is to use the routines getfsent(), getfsspec() or getfsfile().

FILES

/etc/fstab

SEE ALSO

getfsent(3)

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)

BUGS

The *passwd*(1) command won't change the passwords.

NAME

mpxio — multiplexed i/o

SYNOPSIS

#include <sys/mx.h>

#include <sgtty.h>

DESCRIPTION

Data transfers on mpx files (see *mpx*(2)) are multiplexed by imposing a record structure on the io stream. Each record represents data from/to a particular channel or a control or status message associated with a particular channel.

The prototypical data record read from an mpx file is as follows

```
struct input_record {
        short   index;
        short   count;
        short   ccount;
        char    data[];
};
```

where *index* identifies the channel, and *count* specifies the number of characters in *data*. If *count* is zero, *ccount* gives the size of *data*, and the record is a control or status message. Although *count* or *ccount* might be odd, the operating system aligns records on short (i.e. 16−bit) boundaries by skipping bytes when necessary.

Data written to an mpx file must be formatted as an array of record structures defined as follows

```
struct output_record {
        short   index;
        short   count;
        short   ccount;
        char    *data;
};
```

where the data portion of the record is referred to indirectly and the other cells have the same interpretation as in *input_record.*

The control messages listed below may be read from a multiplexed file descriptor. They are presented as two 16-bit integers: the first number is the message code (defined in <*sys/mx.h*>), the second is an optional parameter meaningful only with M_WATCH, M_BLK, and M_SIG.

M_WATCH
: a process 'wants to attach' on this channel. The second parameter is the 16-bit user-id of the process that executed the open.

M_CLOSE
: the channel is closed. This message is generated when the last file descriptor referencing a channel is closed. The *detach* command (see *mpx*(2) should be used in response to this message.

M_EOT
: indicates logical end of file on a channel. If the channel is joined to a typewriter, EOT (control-d) will cause the M_EOT message under the conditions specified in *tty*(4) for end of file. If the channel is attached to a process, M_EOT will be generated whenever the process writes zero bytes on the channel.

M_BLK
: if non-blocking mode has been enabled on an mpx file descriptor *xd* by executing *ioctl(xd, MXNBLK, 0)*, write operations on the file are truncated in the kernel when internal queues become full. This is done on a per-channel basis: the parameter is a count of the number of characters not transferred to the channel

on which M_BLK is received.

M_UBLK is generated for a channel after M_BLK when the internal queues have drained below a threshold.

M_SIG is generated instead of a normal asynchronous signal on channels that are joined to typewriters. The parameter is the signal number.

Two other messages may be generated by the kernel. As with other messages, the first 16-bit quantity is the message code.

M_OPEN is generated in conjunction with 'listener' mode (see *mpx*(2)). The uid of the calling process follows the message code as with M_WATCH. This is followed by a null-terminated string which is the name of the file being opened.

M_IOCTL is generated for a channel connected to a process when that process executes the *ioctl(fd, cmd, &vec)* call on the channel file descriptor. The M_IOCTL code is followed by the *cmd* argument given to *ioctl* followed by the contents of the structure *vec*. It is assumed, not needing a better compromise at this time, that the length of *vec* is determined by *sizeof (struct sgttyb)* as declared in *<sgtty.h>*.

Two control messages are understood by the operating system. M_EOT may be sent through an mpx file to a channel. It is equivalent to propagating a zero-length record through the channel; i.e. the channel is allowed to drain and the process or device at the other end receives a zero-length transfer before data starts flowing through the channel again. M_IOANS can also be sent through a channel to reply to a M_IOCTL. The format is identical to that received from M_IOCTL.

SEE ALSO

mpx(2)

NAME

mtab — mounted file system table

DESCRIPTION

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

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

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

FILES

/etc/mtab

SEE ALSO

mount(8)

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, office, extension, home phone.
initial working directory
program to use as Shell

The name may contain '&', meaning insert the login name. This information is set by the *chfn*(1) command and used by the *finger*(1) command.

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, then */bin/sh* is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

Appropriate precautions must be taken to lock the file against changes if it is to be edited with a text editor; *vipw*(8) does the necessary locking.

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5), chfn(1), finger(1), vipw(8), adduser(8)

BUGS

A binary indexed file format should be available for fast access.

User information (name, office, etc.) should be stored elsewhere.

NAME

plot − graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3), and are interpreted for various devices by commands described in *plot*(1). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an **l, m, n,** or **p** instruction becomes the 'current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in *plot*(3).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See *plot*(1).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

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

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are 'dotted,' 'solid,' 'longdashed,' 'shortdashed,' and 'dotdashed.' Effective only in *plot 4014* and *plot ver.*

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(1). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

4014	space(0, 0, 3120, 3120);
ver	space(0, 0, 2048, 2048);
300, 300s	space(0, 0, 4096, 4096);
450	space(0, 0, 4096, 4096);

SEE ALSO

plot(1), plot(3), graph(1)

NAME

stab — symbol table types

SYNOPSIS

#include <stab.h>

DESCRIPTION

Stab.h defines some values of the n_type field of the symbol table of a.out files. These are the types for permanent symbols (i.e. not local labels, etc.) used by the debugger *sdb*(1) and the Berkeley Pascal compiler *pc*(1). Symbol table entries can be produced by the *.stabs* assembler directive. This allows one to specify a double-quote delimited name, a symbol type, one char and one short of information about the symbol, and an unsigned long (usually an address). To avoid having to produce an explicit label for the address field, the *.stabd* directive can be used to implicitly address the current location. If no name is needed, symbol table entries can be generated using the *.stabn* directive. The loader promises to preserve the order of symbol table entries produced by *.stab* directives. As described in a.out(5), an element of the symbol table consists of the following structure:

```
/*
 * Format of a symbol table entry.
 */
struct nlist {
        union {
                char  *n_name; /* for use when in-core */
                long  n_strx;   /* index into file string table */
        } n_un;
        unsigned char  n_type;   /* type flag */
        char           n_other;  /* unused */
        short          n_desc;   /* see struct desc, below */
        unsigned n_value;        /* address or offset or line */
};
```

The low bits of the n_type field are used to place a symbol into at most one segment, according to the following masks, defined in **<a.out.h>**. A symbol can be in none of these segments by having none of these segment bits set.

```
/*
 * Simple values for n_type.
 */
#define N_UNDF   0x0   /* undefined */
#define N_ABS    0x2   /* absolute */
#define N_TEXT   0x4   /* text */
#define N_DATA   0x6   /* data */
#define N_BSS    0x8   /* bss */

#define N_EXT    01    /* external bit, or'ed in */
```

The n_value field of a symbol is relocated by the linker, *ld*(5) as an address within the appropriate segment. N_value fields of symbols not in any segment are unchanged by the linker. In addition, the linker will discard certain symbols, according to rules of its own, unless the n_type field has one of the following bits set:

```
/*
 * Other permanent symbol table entries have some of the N_STAB bits set.
 * These are given in <stab.h>
 */
#define N_STAB          0xe0/* if any of these bits set, don't discard */
```

This allows up to 112 (7 * 16) symbol types, split between the various segments. Some of these have already been claimed. The symbolic debugger, *sdb*(1), uses the following n_type values:

```
#define N_GSYM   0x20 /* global symbol: name,,0,type,0 */
#define N_FNAME  0x22 /* procedure name (f77 kludge): name,,0 */
#define N_FUN    0x24 /* procedure: name,,0,linenumber,address */
#define N_STSYM  0x26 /* static symbol: name,,0,type,address */
#define N_LCSYM  0x28 /* .lcomm symbol: name,,0,type,address */
#define N_RSYM   0x40 /* register sym: name,,0,type,register */
#define N_SLINE  0x44 /* src line: 0,,0,linenumber,address */
#define N_SSYM   0x60 /* structure elt: name,,0,type,struct_offset */
#define N_SO     0x64 /* source file name: name,,0,0,address */
#define N_LSYM   0x80 /* local sym: name,,0,type,offset */
#define N_SOL    0x84 /* #included file name: name,,0,0,address */
#define N_PSYM   0xa0 /* parameter: name,,0,type,offset */
#define N_ENTRY  0xa4 /* alternate entry: name,linenumber,address */
#define N_LBRAC  0xc0 /* left bracket: 0,,0,nesting level,address */
#define N_RBRAC  0xe0 /* right bracket: 0,,0,nesting level,address */
#define N_BCOMM  0xe2 /* begin common: name,, */
#define N_ECOMM  0xe4 /* end common: name,, */
#define N_ECOML  0xe8 /* end common (local name): ,,address */
#define N_LENG   0xfe /* second stab entry with length information */
```

where the comments give the *sdb* conventional use for *.stab*s and the n_name, n_other, n_desc, and n_value fields of the given n_type. *Sdb* uses the n_desc field to hold a type specifier in the form used by the Portable C Compiler, *cc*(1), in which a base type is qualified in the following structure:

```
struct desc {
        short  q6:2,
               q5:2,
               q4:2,
               q3:2,
               q2:2,
               q1:2,
               basic:4;
};
```

There are four qualifications, with q1 the most significant and q6 the least significant:

0	none
1	pointer
2	function
3	array

The sixteen basic types are assigned as follows:

0	undefined
1	function argument
2	character
3	short
4	int
5	long
6	float
7	double
8	structure
9	union

10	enumeration
11	member of enumeration
12	unsigned character
13	unsigned short
14	unsigned int
15	unsigned long

The Berkeley Pascal compiler, *pc*(1), uses the following n_type value:

```
#define N_PC   0x30    /* global pascal symbol: name,,0,subtype,line */
```

and uses the following subtypes to do type checking across separately compiled files:

1	source file name
2	included file name
3	global label
4	global constant
5	global type
6	global variable
7	global function
8	global procedure
9	external function
10	external procedure

SEE ALSO

as(1), ld(1), sdb(1), a.out(5)

BUGS

Sdb(1) assumes that a symbol of type N_GSYM with name *name* is located at address _*name*.

More basic types are needed.

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.
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\El\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 that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in *termcap* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has "automatic margins" (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. '20', or an integer followed by an '*', i.e. '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a : in a capability it must be escaped in octal as \072. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines 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. (This only works on version 7 systems.)

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&%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 not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining — half bright is not usually an acceptable "standout" mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, this is acceptable, and although it may confuse some programs slightly, it can't be helped.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of *ex*, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasabie 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***x*.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, **is** will be printed before **if**. This is useful where **if** is */usr/lib/tabset/std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap file containing terminal descriptions

SEE ALSO

ex(1), curses(3), termcap(3), tset(1), vi(1), ul(1), more(1)

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

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 *reboot*(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];
        unsigned short  mode;
        char            uid;
        char            gid;
        char            unused1;
        char            size[3];
        long            modtime;
        unsigned short  tapeaddr;
        char            unused2[16];
        unsigned short  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*(8), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (*Getty* will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, /dev.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1)

NAME

ttytype — data base of terminal types by port

SYNOPSIS

/etc/ttytype

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 variable at login time.

SEE ALSO

tset(1), login(1)

BUGS

Some lines are merely known as "dialup" or "plugboard".

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 struct _physadr { int r[1]; } *physadr;
typedef long            daddr_t;
typedef char *          caddr_t;
typedef unsigned short  ino_t;
typedef int             swblk_t;
typedef int             size_t;
typedef long            time_t;
typedef long            label_t[14];
typedef short           dev_t;
typedef long            off_t;

typedef unsigned char   u_char;
typedef unsigned short  u_short;
typedef unsigned int    u_int;
typedef unsigned long   u_long;

/* major part of a device */
#define major(x)  ((int)(((unsigned)(x)>>8)&0377))

/* minor part of a device */
#define minor(x)  ((int)((x)&0377))

/* make a device number */
#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 UNIX. The file is a sequence of entries with the following structure declared in the include file:

```
/*
 * Structure of utmp and wtmp files.
 *
 * Assuming the number 8 is unwise.
 */
struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user id */
        long    ut_time;                /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(2).

The *wtmp* file records all logins and logouts. Its format is exactly like *utmp* except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names '|' and '}' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by *ac*(8).

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

login(1), init(8), who(1), ac(8)

NAME

uuencode − format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1)* consist of a header line, followed by a number of body lines, and a trailer line. *Uudecode(1)* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin ". The word *begin* is followed by a mode (in octal), and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

SEE ALSO

uuencode(1), uudecode(1), uusend(1), uucp(1), mail(1)

NAME

vfont − font formats for the Benson-Varian or Versatec

SYNOPSIS

/usr/lib/vfont/*

DESCRIPTION

The fonts for the printer/plotters have the following format. Each file contains a header, an array of 256 character description structures, and then the bit maps for the characters themselves. The header has the following format:

```
struct header {
        short           magic;
        unsigned short  size;
        short           maxx;
        short           maxy;
        short           xtnd;
} header;
```

The *magic* number is 0436 (octal). The *maxx*, *maxy*, and *xtnd* fields are not used at the current time. *Maxx* and *maxy* are intended to be the maximum horizontal and vertical size of any glyph in the font, in raster lines. The *size* is the size of the bit maps for the characters in bytes. Before the maps for the characters is an array of 256 structures for each of the possible characters in the font. Each element of the array has the form:

```
struct dispatch {
        unsigned short  addr;
        short           nbytes;
        char            up;
        char            down;
        char            left;
        char            right;
        short           width;
};
```

The *nbytes* field is nonzero for characters which actually exist. For such characters, the *addr* field is an offset into the rest of the file where the data for that character begins. There are *up+down* rows of data for each character, each of which has *left+right* bits, rounded up to a number of bytes. The *width* field is not used by vcat, although it is used by *vwidth*(1) to make width tables for *troff*. It represents the logical width of the glyph, in raster lines, and shows where the base point of the next glyph would be.

FILES

/usr/lib/vfont/*

SEE ALSO

troff(1), pti(1), vpr(1), vtroff(1), vwidth(1), vfontinfo(1), fed(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*(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by *ac*(1).

FILES

/usr/adm/wtmp

SEE ALSO

utmp(5), login(1), init(8), ac(1), who(1)

NAME
aardvark — yet another exploration game

SYNOPSIS
/usr/games/aardvark

DESCRIPTION
Aardvark is yet another computer fantasy simulation game of the adventure/zork genre. This one is written in DDL (Dungeon Definition Language) and is intended primarily as an example of how to write a dungeon in DDL.

FILES
/usr/games/lib/ddlrun ddl interpreter
/usr/games/lib/aardvarkinternal form of aardvark dungeon

AUTHOR
Mike Urban, UCLA

BUGS

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.

SEE ALSO

dd(1)

NAME

boggle – play the game of boggle

SYNOPSIS

/usr/games/boggle [+] [++]

DESCRIPTION

This program is intended for people wishing to sharpen their skills at Boggle (TM Parker Bros.). If you invoke the program with 4 arguments of 4 letters each, (*e.g.* "**boggle appl epie moth erhd**") the program forms the obvious Boggle grid and lists all the words from **/usr/dict/words** found therein. If you invoke the program without arguments, it will generate a board for you, let you enter words for 3 minutes, and then tell you how well you did relative to **/usr/dict/words**.

The object of Boggle is to find, within 3 minutes, as many words as possible in a 4 by 4 grid of letters. Words may be formed from any sequence of 3 or more adjacent letters in the grid. The letters may join horizontally, vertically, or diagonally. However, no position in the grid may be used more than once within any one word. In competitive play amongst humans, each player is given credit for those of his words which no other player has found.

In interactive play, enter your words separated by spaces, tabs, or newlines. A bell will ring when there is 2:00, 1:00, 0:10, 0:02, 0:01, and 0:00 time left. You may complete any word started before the expiration of time. You can surrender before time is up by hitting 'break'. While entering words, your erase character is only effective within the current word and your line kill character is ignored.

Advanced players may wish to invoke the program with 1 or 2 +'s as the first argument. The first + removes the restriction that positions can only be used once in each word. The second + causes a position to be considered adjacent to itself as well as its (up to) 8 neighbors.

NAME

chase — Try to escape to killer robots

SYNOPSIS

/usr/games/chase [*nrobots*] [*nfences*]

DESCRIPTION

The object of the game chase is to move around inside of the box on the screen without getting eaten by the robots chasing and without running into anything.

If a robot runs into another robot while chasing you, they crash and leave a junk heap. If a robot runs into a fence, it is destroyed.

If you can survive until all the robots are destroyed, you have won!

If you do not specify either *nrobots* or *nfences,* chase will prompt you for them.

BUGS

NAME

chess − the game of chess

SYNOPSIS

/usr/games/chess

DESCRIPTION

Chess is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '+' is used to specify check; 'o-o' and 'o-o-o' specify castling. To play black, type 'first'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

FILES

/usr/lib/book	opening 'book'

DIAGNOSTICS

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

WARNING

Over-use of this program will cause it to go away.

BUGS

Pawns may be promoted only to queens.

NAME

ching, fortune — the book of changes and other cookies

SYNOPSIS

/usr/games/ching [hexagram]

/usr/games/fortune

DESCRIPTION

The *I Ching* or *Book of Changes* is an ancient Chinese oracle that has been in use for centuries as a source of wisdom and advice.

The text of the *oracle* (as it is sometimes known) consists of sixty-four *hexagrams*, each symbolized by a particular arrangement of six straight (— — —) and broken (— —) lines. These lines have values ranging from six through nine, with the even values indicating the broken lines.

Each hexagram consists of two major sections. The **Judgement** relates specifically to the matter at hand (E.g., "It furthers one to have somewhere to go.") while the **Image** describes the general attributes of the hexagram and how they apply to one's own life ("Thus the superior man makes himself strong and untiring.").

When any of the lines have the values six or nine, they are moving lines; for each there is an appended judgement which becomes significant. Furthermore, the moving lines are inherently unstable and change into their opposites; a second hexagram (and thus an additional judgement) is formed.

Normally, one consults the oracle by fixing the desired question firmly in mind and then casting a set of changes (lines) using yarrow—stalks or tossed coins. The resulting hexagram will be the answer to the question.

Using an algorithm suggested by S. C. Johnson, the Unix *oracle* simply reads a question from the standard input (up to an EOF) and hashes the individual characters in combination with the time of day, process id and any other magic numbers which happen to be lying around the system. The resulting value is used as the seed of a random number generator which drives a simulated coin—toss divination. The answer is then piped through **nroff** for formatting and will appear on the standard output.

For those who wish to remain steadfast in the old traditions, the oracle will also accept the results of a personal divination using, for example, coins. To do this, cast the change and then type the resulting line values as an argument.

The impatient modern may prefer to settle for Chinese cookies; try *fortune.*

SEE ALSO

It furthers one to see the great man.

DIAGNOSTICS

The great prince issues commands,
Founds states, vests families with fiefs.
Inferior people should not be employed.

BUGS

Waiting in the mud
Brings about the arrival of the enemy.

If one is not extremely careful,
Somebody may come up from behind and strike him.
Misfortune.

NAME

cribbage — the card game cribbage

SYNOPSIS

/usr/games/cribbage

DESCRIPTION

Cribbage plays the card game cribbage, with the program playing one hand and the user the other. For a complete description of the rules of cribbage, refer to *According to Hoyle.*

Cribbage first asks the player whether he wishes to play a short game ("once around", to 61) or a long game ("twice around", to 121). A response of 's' will result in a short game, any other response will play a long game.

At the start of the first game, the program asks the player to cut the deck to determine who gets the first crib. The user should respond with a number between 0 and 51, indicating how many cards down the deck is to be cut. The player who cuts the lower ranked card gets the first crib. If more than one game is played, the loser of the previous game gets the first crib in the current game.

For each hand, the program first prints the player's hand, whose crib it is, and then asks the player to discard two cards into the crib. The cards are prompted for one per line, and are typed as explained below.

After discarding, the program cuts the deck (if it is the player's crib) or asks the player to cut the deck (if it's its crib); in the later case, the appropriate response is a number from 0 to 39 indicating how far down the remaining 40 cards are to be cut.

After cutting the deck, play starts with the non-dealer (the person who doesn't have the crib) leading the first card. Play continues, as per cribbage, until all cards are exhausted. The program keeps track of the scoring of all points and the total of the cards on the table.

After play, the hands are scored. The program requests the player to score his hand (and the crib, if it is his) by printing out the appropriate cards (and the cut card enclosed in brackets). Play continues until one player reaches the game limit (61 or 121).

A carriage return when a numeric input is expected is equivalent to typing the lowest legal value; when cutting the deck this is equivalent to choosing the top card.

Cards are specified as rank followed by suit. The ranks may be specified as one of: 'a', '2', '3', '4', '5', '6', '7', '8', '9', 't', 'j', 'q', and 'k', or alternatively, one of: "ace", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten", "jack", "queen", and "king". Suits may be specified as: 's', 'h', 'd', and 'c', or alternatively as: "spades", "hearts", "diamonds", and "clubs". A card may be specified as: <rank> " " <suit>, or: <rank> " of " <suit>. If the single letter rank and suit designations are used, the space separating the suit and rank may be left out. Also, if only one card of the desired rank is playable, typing the rank is sufficient. For example, if your hand was "2H, 4D, 5C, 6H, JC, KD" and it was desired to discard the king of diamonds, any of the following could be typed: "k", "king", "kd", "k d", "k of d", "king d", "king of d", "k diamonds", "k of diamonds", "king diamonds", or "king of diamonds".

FILES

/usr/games/cribbage

AUTHOR

Earl T. Cohen

BUGS

NAME

doctor — interact with a psychoanalyst

SYNOPSIS

/usr/games/doctor

DESCRIPTION

Doctor is a lisp-language version of the legendary ELIZA program of Joseph Weizenbaum. This script "simulates" a Rogerian psychoanalyst. Type in lower case, and when you get tired or bored, type your interrupt character (either control-C or Rubout). Remember to type two carriage returns when you want it to answer.

In order to run this you must have a Franz Lisp system in /usr/ucb/lisp.

AUTHORS

Adapted for Lisp by Jon L White, moved to Franz by John Foderaro, from an original script by Joseph Weizenbaum.

BUG

It shows how impressed people were willing to be way back then.

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 [**—**] [**—wslao**]

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.

—o Choose from an alternate list of adages, often used for portentially offensive ones.

—a Choose from either list of adages.

Mail suggestions for new fortunes to "fortune".

FILES

/usr/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 in a graphic fashion.

FILES

/usr/dict/words On-line word list

AUTHOR

Modified for terminal graphics from the original source from BTL by Michael Toy.

BUGS

NAME

mille – play Mille Bournes

SYNOPSIS

/usr/games/mille [file]

DESCRIPTION

Mille plays a two-handed game reminiscent of the Parker Brother's game of Mille Bournes with you. The rules are described below. If a file name is given on the command line, the game saved in that file is started.

When a game is started up, the bottom of the score window will contain a list of commands. They are:

P Pick a card from the deck. This card is placed in the 'P' slot in your hand.

D Discard a card from your hand. To indicate which card, type the number of the card in the hand (or "P" for the just-picked card) followed by a <RETURN> or <SPACE>. The <RETURN or <SPACE> is required to allow recovery from typos which can be very expensive, like discarding safeties.

U Use a card. The card is again indicated by its number, followed by a <RETURN> or <SPACE>.

O Toggle ordering the hand. By default off, if turned on it will sort the cards in your hand appropriately. This is not recommended for the impatient on slow terminals.

Q Quit the game. This will ask for confirmation, just to be sure. Hitting <DELETE> (or <RUBOUT>) is equivalent.

S Save the game in a file. If the game was started from a file, you will be given an opportunity to save it on the same file. If you don't wish to, or you did not start from a file, you will be asked for the file name. If you type a <RETURN> without a name, the save will be terminated and the game resumed.

R Redraw the screen from scratch. The command ^L (control 'L') will also work.

W Toggle window type. This switches the score window between the startup window (with all the command names) and the end-of-game window. Using the end-of-game window saves time by eliminating the switch at the end of the game to show the final score. Recommended for hackers and other miscreants.

If you make a mistake, an error message will be printed on the last line of the score window, and a bell will beep.

At the end of each hand or game, you will be asked if you wish to play another. If not, it will ask you if you want to save the game. If you do, and the save is unsuccessful, play will be resumed as if you had said you wanted to play another hand/game. This allows you to use the "S" command to reattempt the save.

AUTHOR

Ken Arnold
(The game itself is a product of Parker Brothers, Inc.)

SEE ALSO

curses(3), *Screen Updating and Cursor Movement Optimization: A Library Package*, Ken Arnold

CARDS

Here is some useful information. The number in parentheses after the card name is the number of that card in the deck:

Hazard	Repair	Safety
Out of Gas (2)	Gasoline (6)	Extra Tank (1)
Flat Tire (2)	Spare Tire (6)	Puncture Proof (1)
Accident (2)	Repairs (6)	Driving Ace (1)
Stop (4)	Go (14)	Right of Way (1)
Speed Limit (3)	End of Limit (6)	

25 – (10), 50 – (10), 75 – (10), 100 – (12), 200 – (4)

RULES

Object: The point of game is to get a total of 5000 points in several hands. Each hand is a race to put down exactly 700 miles before your opponent does. Beyond the points gained by putting down milestones, there are several other ways of making points.

Overview: The game is played with a deck of 101 cards. *Distance* cards represent a number of miles traveled. They come in denominations of 25, 50, 75, 100, and 200. When one is played, it adds that many miles to the player's trip so far this hand. *Hazard* cards are used to prevent your opponent from putting down Distance cards. They can only be played if your opponent has a *Go* card on top of the Battle pile. The cards are *Out of Gas*, *Accident*, *Flat Tire*, *Speed Limit*, and *Stop*. *Remedy* cards fix problems caused by Hazard cards played on you by your opponent. The cards are *Gasoline*, *Repairs*, *Spare Tire*, *End of Limit*, and *Go*. *Safety* cards prevent your opponent from putting specific Hazard cards on you in the first place. They are *Extra Tank*, *Driving Ace*, *Puncture Proof*, and *Right of Way*, and there are only one of each in the deck.

Board Layout: The board is split into several areas. From top to bottom, they are: **SAFETY AREA** (unlabeled): This is where the safeties will be placed as they are played. **HAND**: These are the cards in your hand. **BATTLE**: This is the Battle pile. All the Hazard and Remedy Cards are played here, except the *Speed Limit* and *End of Limit* cards. Only the top card is displayed, as it is the only effective one. **SPEED**: The Speed pile. The *Speed Limit* and *End of Limit* cards are played here to control the speed at which the player is allowed to put down miles. **MILEAGE**: Miles are placed here. The total of the numbers shown here is the distance traveled so far.

Play: The first pick alternates between the two players. Each turn usually starts with a pick from the deck. The player then plays a card, or if this is not possible or desirable, discards one. Normally, a play or discard of a single card constitutes a turn. If the card played is a safety, however, the same player takes another turn immediately.

This repeats until one of the players reaches 700 points or the deck runs out. If someone reaces 700, they have the option of going for an *Extension*, which means that the play continues until someone reaches 1000 miles.

Hazard and Remedy Cards: Hazard Cards are played on your opponent's Battle and Speed piles. Remedy Cards are used for undoing the effects of your opponent's nastyness.

Go (Green Light) must be the top card on your Battle pile for you to play any mileage, unless you have played the *Right of Way* card (see below).

Stop is played on your opponent's *Go* card to prevent them from playing mileage until they play a *Go* card.

Speed Limit is played on your opponent's Speed pile. Until they play an *End of Limit* they can only play 25 or 50 mile cards, presuming their *Go* card allows them to do even that.

End of Limit is played on your Speed pile to nullify a *Speed Limit* played by your opponent.

Out of Gas is played on your opponent's *Go* card. They must then play a *Gasoline* card, and then a *Go* card before they can play any more mileage.

Flat Tire is played on your opponent's *Go* card. They must then play a *Spare Tire* card, and then a *Go* card before they can play any more mileage.

Accident is played on your opponent's *Go* card. They must then play a *Repairs* card, and then a *Go* card before they can play any more mileage.

Safety Cards: Safety cards prevent your opponent from playing the corresponding Hazard cards on you for the rest of the hand. It cancels an attack in progress, and *always entitles the player to an extra turn.*

Right of Way prevents your opponent from playing both *Stop* and *Speed Limit* cards on you. It also acts as a permanent *Go* card for the rest of the hand, so you can play mileage as long as there is not a Hazard card on top of your Battle pile. In this case only, your opponent can play Hazard cards directly on a Remedy card besides a Go card.

Extra Tank When played, your opponent cannot play an *Out of Gas* on your Battle Pile.

Puncture Proof When played, your opponent cannot play a *Flat Tire* on your Battle Pile.

Driving Ace When played, your opponent cannot play an *Accident* on your Battle Pile.

Distance Cards: Distance cards are played when you have a *Go* card on your Battle pile, or a Right of Way in your Safety area and are not stopped by a Hazard Card. They can be played in any combination that totals exactly 700 miles, except that *you cannot play more than two 200 mile cards in one hand.* A hand ends whenever one player gets exactly 700 miles or the deck runs out. In that case, play continues until neither someone reaches 700, or neither player can use any cards in their hand. If the trip is completed after the deck runs out, this is called *Delayed Action.*

Coup Fourré: This is a French fencing term for a counter-thrust move as part of a parry to an opponents attack. In Mille Bournes, it is used as follows: If an opponent plays a Hazard card, and you have the corresponding Safety in your hand, you play it immediately, even *before* you draw. This immediately removes the Hazard card from your Battle pile, and protects you from that card for the rest of the game. This gives you more points (see "Scoring" below).

Scoring: Scores are totaled at the end of each hand, whether or not anyone completed the trip. The terms used in the Score window have the following meanings:

Milestones Played: Each player scores as many miles as they played before the trip ended.

Each Safety: 100 points for each safety in the Safety area.

All 4 Safeties: 300 points if all four safeties are played.

Each Coup Fouré: 300 points for each Coup Fouré accomplished.

The following bonus scores can apply only to the winning player.

Trip Completed: 400 points bonus for completing the trip to 700 or 1000.

Safe Trip: 300 points bonus for completing the trip without using any 200 mile cards.

Delayed Action: 300 points bonus for finishing after the deck was exhausted.

Extension: 200 points bonus for completing a 1000 mile trip.

Shut-Out: 500 points bonus for completing the trip before your opponent played any mileage cards.

Running totals are also kept for the current score for each player for the hand (**Hand Total**), the game (**Overall Total**), and number of games won (**Games**).

NAME

monop — Monopoly game

SYNOPSIS

/usr/games/monop [file]

DESCRIPTION

Monop is reminiscent of the Parker Brother's game Monopoly, and monitors a game between 1 to 9 users. It is assumed that the rules of Monopoly are known. The game follows the standard rules, with the exception that, if a property would go up for auction and there are only two solvent players, no auction is held and the property remains unowned.

The game, in effect, lends the player money, so it is possible to buy something which you cannot afford. However, as soon as a person goes into debt, he must "fix the problem", *i.e.*, make himself solvent, before play can continue. If this is not possible, the player's property reverts to his debtee, either a player or the bank. A player can resign at any time to any person or the bank, which puts the property back on the board, unowned.

Any time that the response to a question is a *string*, e.g., a name, place or person, you can type '?' to get a list of valid answers. It is not possible to input a negative number, nor is it ever necessary.

A Summary of Commands:

quit: quit game: This allows you to quit the game. It asks you if you're sure.

print: print board: This prints out the current board. The columns have the following meanings (column headings are the same for the **where**, **own holdings**, and **holdings** commands):

- Name — The first ten characters of the name of the square
- Own — The *number* of the owner of the property.
- Price — The cost of the property (if any)
- Mg — This field has a '*' in it if the property is mortgaged
- # — If the property is a Utility or Railroad, this is the number of such owned by the owner. If the property is land, this is the number of houses on it.
- Rent — Current rent on the property. If it is not owned, there is no rent.

where: where players are: Tells you where all the players are. A '*' indicates the current player.

own holdings:
List your own holdings, *i.e.*, money, get-out-of-jail-free cards, and property.

holdings: holdings list: Look at anyone's holdings. It will ask you whose holdings you wish to look at. When you are finished, type "done".

shell: shell escape: Escape to a shell. When the shell dies, the program continues where you left off.

mortgage: mortgage property: Sets up a list of mortgageable property, and asks which you wish to mortgage.

unmortgage:
unmortgage property: Unmortgage mortgaged property.

buy: buy houses: Sets up a list of monopolies on which you can buy houses. If there is more than one, it asks you which you want to buy for. It then asks you how many for each piece of property, giving the current amount in parentheses after the property name. If you build in an unbalanced manner (a disparity of more than one house within the same monopoly), it asks you to re-input things.

sell: sell houses: Sets up a list of monopolies from which you can sell houses. it operates in an analogous manner to *buy*

card: card for jail: Use a get-out-of-jail-free card to get out of jail. If you're not in jail, or you don't have one, it tells you so.

pay: pay for jail: Pay $50 to get out of jail, from whence you are put on Just Visiting. Difficult to do if you're not there.

trade: This allows you to trade with another player. It asks you whom you wish to trade with, and then asks you what each wishes to give up. You can get a summary at the end, and, in all cases, it asks for confirmation of the trade before doing it.

resign: Resign to another player or the bank. If you resign to the bank, all property reverts to its virgin state, and get-out-of-jail free cards revert to the deck.

save: save game: Save the current game in a file for later play. You can continue play after saving, either by adding the file in which you saved the game after the *monop* command, or by using the *restore* command (see below). It will ask you which file you wish to save it in, and, if the file exists, confirm that you wish to overwrite it.

restore: restore game: Read in a previously saved game from a file. It leaves the file intact.

roll: Roll the dice and move forward to your new location. If you simply hit the <RETURN> key instead of a command, it is the same as typing *roll*.

AUTHOR

Ken Arnold

FILES

/usr/games/lib/cards.pck   Chance and Community Chest cards

BUGS

No command can be given an argument instead of a response to a query.

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. Punctuation is added to make the output sound well when played through *speak*(1).

SEE ALSO

speak(1)

NAME

quiz — test your knowledge

SYNOPSIS

/usr/games/quiz [**−i** file] [**−t**] [category1 category2]

DESCRIPTION

Quiz gives associative knowledge tests on various subjects. It asks items chosen from *category1* and expects answers from *category2*. If no categories are specified, *quiz* gives instructions and lists the available categories.

Quiz tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, *quiz* reports a score and terminates.

The **−t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The **−i** flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

```
line      = category newline | category ':' line
category  = alternate | category '|' alternate
alternate = empty | alternate primary
primary   = character | '[' category ']' | option
option    = '{' category '}'
```

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash '\' is used as with *sh*(1) to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, *quiz* will refrain from asking it.

FILES

/usr/games/quiz.k/*

BUGS

The construct 'a|ab' doesn't work in an information file. Use 'a{b}'.

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

rogue — Exploring The Dungeons of Doom

SYNOPSIS

rogue [*save_file*]

DESCRIPTION

Rogue is a computer fantasy game with a new twist. It is crt oriented and the object of the game is to survive the attacks of various monsters and get a lot of gold, rather than the puzzle solving orientation of most computer fantasy games.

To get started you really only need to know two commands. The command ? will give you a list of the available commands and the command / will identify the things you see on the screen.

To win the game (as opposed to merely playing to beat other people high scores) you must locate the Amulet of Yendor which is somewhere below the 20th level of the dungeon and get it out. Nobody has achieved this yet and if somebody does, they will probably go down in history as a hero among heros.

When the game ends, either by your death, when you quit, or if you (by some miracle) manage to win, *rogue* will give you alist of the top-ten scorers. The scoring is based entirely upon how much gold you get. There is a 10% penalty for getting yourself killed.

For more detailed directions, read the document *A Guide to the Dungeons of Doom.*

FILES

/usr/games/lib/rogue_roll Score file
~/rogue.sav Default save file

SEE ALSO

Michael C. Toy, *A guide to the Dungeons of Doom*

BUGS

Probably infinite. Currently known bugs are: Sometimes you are still hungry even after you eat food and sometimes you get a monster on the screen in reverse video which may or may not cause a core dump.

NAME

snake, snscore — display chase game

SYNOPSIS

/usr/games/snake [**−w***n*] [**−l***n*]
/usr/games/snscore

DESCRIPTION

Snake is a display-based game which must be played on a CRT terminal from among those supported by vi(1). The object of the game is to make as much money as possible without getting eaten by the snake. The **−l** and **−w** options allow you to specify the length and width of the field. By default the entire screen (except for the last column) is used.

You are represented on the screen by an I. The snake is 6 squares long and is represented by S's. The money is $, and an exit is #. Your score is posted in the upper left hand corner.

You can move around using the same conventions as vi(1), the h, j, k, and l keys work, as do the arrow keys. Other possibilities include:

sefc These keys are like hjkl but form a directed pad around the d key.

HJKL These keys move you all the way in the indicated direction to the same row or column as the money. This does *not* let you jump away from the snake, but rather saves you from having to type a key repeatedly. The snake still gets all his turns.

SEFC Likewise for the upper case versions on the left.

ATPB These keys move you to the four edges of the screen. Their position on the keyboard is the mnemonic, e.g. P is at the far right of the keyboard.

x This lets you quit the game at any time.

p Points in a direction you might want to go.

w Space warp to get out of tight squeezes, at a price.

! Shell escape

^Z Suspend the snake game, on systems which support it. Otherwise an interactive shell is started up.

To earn money, move to the same square the money is on. A new $ will appear when you earn the current one. As you get richer, the snake gets hungrier. To leave the game, move to the exit (#).

A record is kept of the personal best score of each player. Scores are only counted if you leave at the exit, getting eaten by the snake is worth nothing.

As in pinball, matching the last digit of your score to the number which appears after the game is worth a bonus.

To see who wastes time playing snake, run */usr/games/snscore* .

FILES

/usr/games/lib/snakerawscores	database of personal bests
/usr/games/lib/snake.log	log of games played
/usr/games/busy	program to determine if system too busy

BUGS

When playing on a small screen, it's hard to tell when you hit the edge of the screen.

The scoring function takes into account the size of the screen. A perfect function to do this equitably has not been devised.

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
cloak **up**/**do**wn
computer request; ...
destruct
help
lrscan
phasers **a**utomatic amount
phasers **m**anual amt1 course1 spread1 ...
torpedo course [yes] angle/**no**
ram course distance
shell
srscan [yes/**no**]
status
undock
warp warp_factor

capture
damages
dock
impulse course distance
move course distance
rest time
shields up/**do**wn
terminate yes/**no**
visual course

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 is will grow longer, the actual amount longer depends on which digit it was that you ate. 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

Brian Horn (cithep!bdh) showed me a *TOPS-20* program on the DEC-2136 machine called *WORM*, and suggested that I write a similar program that would run under *Unix*. I did, and no apologies.

-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

SEE ALSO

Snails, by Karl Heuer

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 on a VAX.

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.

There should be an option to have the worms eat *Pink Floyd* lyrics.

NAME

wump — the game of hunt-the-wumpus

SYNOPSIS

/usr/games/wump

DESCRIPTION

Wump plays the game of 'Hunt the Wumpus.' A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company, 2,* 2 (November 1973).

BUGS

It will never replace Space War.

NAME

zork — the game of dungeon

SYNOPSIS

/usr/games/zork

DESCRIPTION

Dungeon is a computer fantasy simulation based on Adventure and on Dungeons & Dragons, originally written by Lebling, Blank, and Anderson of MIT. In it you explore a dungeon made up of various rooms, caves, rivers, and so on. The object of the game is to collect as much treasure as possible and stow it safely in the trophy case (and, of course, to stay alive.)

Figuring out the rules is part of the game, but if you are stuck, you should start off with "open mailbox", "take leaflet", and then "read leaflet". Additional useful commands that are not documented include:

quit	(to end the game)
!cmd	(the usual shell escape convention)
>	(to save a game)
<	(to restore a game)

FILES

/usr/games/lib/d*

BUGS

We don't have the source, only a pdp-11 binary that has been severely munged to get it to work on V7 Unix. (The original binary was for RSX-11, which was patched for V6, and then patched for V7.)

NAME

ascii — map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
010 bs	011 ht	012 nl	013 vt	014 np	015 cr	016 so	017 si
020 dle	021 dc1	022 dc2	023 dc3	024 dc4	025 nak	026 syn	027 etb
030 can	031 em	032 sub	033 esc	034 fs	035 gs	036 rs	037 us
040 sp	041 !	042 "	043 #	044 $	045 %	046 &	047 '
050 (	051)	052 *	053 +	054 ,	055 −	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 del

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

FILES

/usr/pub/ascii

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

ciplus	⊕	*||*	‖	*square*	□
citimes	⊗	*langle*	⟨	*circle*	○
wig	∼	*rangle*	⟩	*blot*	■
-wig	≃	*hbar*	ℏ	*bullet*	●
>wig	≳	*ppd*	⊥	*prop*	∝
<wig	≲	*<->*	⟷	*empty*	∅
=wig	≅	*<=>*	⟺	*member*	∈
star	*	*|<*	≮	*nomem*	∉
bigstar	✱	*|>*	≯	*cup*	∪
=dot	≐	*ang*	∠	*cap*	∩
orsign	∨	*rang*	∟	*incl*	⊑
andsign	∧	*3dot*	⋮	*subset*	⊂
=del	≜	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*	¼	*!subset*	⊆
oppE	∃	*3quarter*	¾	*!supset*	⊇
angstrom	Å	*degree*	°		

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	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ϵ	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	¬	_
partial	∂	]	integral	∫	^			

SEE ALSO

greek(1)
troff(1)

NAME

hier − file system hierarchy

DESCRIPTION

The following outline gives a quick tour through a representative directory hierarchy.

```
/         root
/vmunix
          the kernel binary (UNIX itself)
/lost+found
          directory for connecting detached files for fsck(5)
/dev/     devices (4)
          console
                    main console, tty(4)
          tty*      terminals, tty(4)
          rp*       disks, hp(4)
          rrp*      raw disks, hp(4)
          up*       UNIBUS disks up(4)
          ...
/bin/     utility programs, cf /usr/bin/ (1)
          as        assembler
          cc        C compiler executive, cf /lib/ccom, /lib/cpp, /lib/c2
          csh       C shell
          ...
/lib/     object libraries and other stuff, cf /usr/lib/
          libc.a    system calls, standard I/O, etc. (2,3,3S)
          ...
          ccom      C compiler proper
          cpp       C preprocessor
          c2        C code improver
          ...
/etc/     essential data and maintenance utilities; sect (8)
          dump      dump program dump(8)
          passwd    password file, passwd(5)
          group     group file, group(5)
          motd      message of the day, login(1)
          termcap
                    description of terminal capabilities, termcap(5)
          ttytype   table of what kind of terminal is on each port, ttytype(5)
          mtab      mounted file table, mtab(5)
          dumpdates
                    dump history, dump(8)
          fstab     file system configurtion table fstab(5)
          ttys      properties of terminals, ttys(5)
          getty     part of login, getty(8)
          init      the parent of all processes, init.vm(8)
          rc        shell program to bring the system up
          cron      the clock daemon, cron(8)
          mount     mount(8)
          wall      wall(8)
          ...
/tmp/     temporary files, usually on a fast device, cf /usr/tmp/
          e*        used by ed(1)
          ctm*      used by cc(1)
```

```
        ...
/usr/   general-pupose directory, usually a mounted file system
        adm/    administrative information
                wtmp    login history, utmp(5)
                messages
                        hardware error messages
                tracct  phototypesetter accounting, troff(1)
                lpacct  line printer accounting lpr(1)
                vaacct, vpacct
                        varian and versatec accounting vpr(1), vtroff(1), vpac(1)
/usr    /bin
        utility programs, to keep /bin/ small
        tmp/    temporaries, to keep /tmp/ small
                stm*    used by sort(1)
                raster  used by plot(1)
        dict/   word lists, etc.
                words   principal word list, used by look(1)
                spellhist
                        history file for spell(1)
        games/
                hangman
                lib/    library of stuff for the games
                        quiz.k/ what quiz(6) knows
                                index   category index
                                africa  countries and capitals
                                ...
                        ...
                ...
        include/
                standard #include files
                a.out.h object file layout, a.out(5)
                stdio.h standard I/O, stdio(3)
                math.h  (3M)
                ...
                sys/    system-defined layouts, cf /usr/sys/h
        lib/    object libraries and stuff, to keep /lib/ small
                atrun   scheduler for at(1)
                lint/   utility files for lint
                        lint[12]
                                subprocesses for lint(1)
                        llib-lc dummy declarations for /lib/libc.a, used by lint(1)
                        llib-lm dummy declarations for /lib/libc.m
                        ...
                struct/ passes of struct(1)
                ...
                tmac/   macros for troff(1)
                        tmac.an
                                macros for man(7)
                        tmac.s  macros for ms(7)
                        ...
                font/   fonts for troff(1)
                        ftR     Times Roman
```

```
                ftB     Times Bold
                ...
          uucp/ programs and data for uucp(1)
                L.sys   remote system names and numbers
                uucico  the real copy program
                ...
          units conversion tables for units(1)
          eign  list of English words to be ignored by ptx(1)
/usr/  man/
    volume 1 of this manual, man(1)
          man0/ general
                intro   introduction to volume 1, ms(7) format
                xx      template for manual page
          man1/ chapter 1
                as.1
                mount.1m
                ...
          ...
          cat1/ preformatted pages for section 1
          ...
    msgs/ messages, cf msgs(1)
          bounds highest and lowest message
    new/  binaries of new versions of programs
    preserve/
          editor temporaries preserved here after crashes/hangups
    public/ binaries of user programs - write permission to everyone
    spool/ delayed execution files
          at/   used by at(1)
          lpd/  used by lpr(1)
                lock    present when line printer is active
                cf*     copy of file to be printed, if necessary
                df*     daemon control file, lpd(8)
                tf*     transient control file, while lpr is working
          uucp/ work files and staging area for uucp(1)
                LOGFILE
                        summary log
                LOG.*   log file for one transaction
          mail/ mailboxes for mail(1)
                name    mail file for user name
                name.lock
                        lock file while name is receiving mail
          secretmail/
                like mail/
          uucp/ work files and staging area for uucp(1)
                LOGFILE
                        summary log
                LOG.*   log file for one transaction
    wd    initial working directory of a user, typically wd is the user's login name
          .profile set environment for sh(1), environ(5)
          .project
                what you are doing (used by ( finger(1) )
          .cshrc startup file for csh(1)
```

```
                .exrc   startup file for ex(1)
                .plan   what your short-term plans are (used by finger(1) )
                .netrc  startup file for net(1)
                .msgsrc
                        startup file for msgs(1)
                .mailrc startup file for mail(1)
                calendar
                        user's datebook for calendar(1)
        doc/    papers, mostly in volume 2 of this manual, typically in ms(7) format
                as/     assembler manual
                c       C manual
                ...
/usr/   src/
        source programs for utilities, etc.
        cmd/    source of commands
                as/     assembler
                ar.c    source for ar(1)
                ...
                troff/  source for nroff and troff(1)
                        font/   source for font tables, /usr/lib/font/
                                ftR.c   Roman
                                ...
                        term/   terminal characteristics tables, /usr/lib/term/
                                tab300.c
                                        DASI 300
                                ...
                        ...
        games/  source for /usr/games
        libc/   source for functions in /lib/libc.a
                crt/    C runtime support
                csu/    startup and wrapup routines needed with every C program
                        crt0.s  regular startup
                        mcrt0.s modified startup for cc -p
                sys/    system calls (2)
                        access.s
                        alarm.s
                        ...
                stdio/  standard I/O functions (3S)
                        fgets.c
                        fopen.c
                        ...
                gen/    other functions in (3)
                        abs.c
                        ...
        local/  source which isn't normally distributed
        new/    source for new versions of commands and library routines
        old/    source for old versions of commands and library routines
        sys/    system source
                h/      header (include) files
                        acct.h  acct(5)
                        stat.h  stat(2)
                        ...
```

sys/ system source proper
main.c
pipe.c
sysent.c
system entry points

ucb/ binaries of programs developed at UCB
...
edit editor for beginners
ex command editor for experienced users
...
mail mail reading/sending subsystem
man on line documentation
...
pi Pascal translator
px Pascal interpreter
...
vi visual editor

SEE ALSO

ls(1), apropos(1), whatis(1), whereis(1), finger(1), which (1), ncheck(8), find(1), grep(1)

BUGS

The position of files is subject to change without notice.

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. A skeleton page may be found in the file /usr/man/man0/xx.

Any text argument *t* may be zero to six words. Quotes may be used to include blanks in a 'word'. If *text* is empty, the special treatment is applied to the next input line with text to be printed. In this way .I may be used to italicize a whole line, or .SM followed by .B to make small bold letters.

A prevailing indent distance is remembered between successive indented paragraphs, and is reset to default value upon reaching a non-indented paragraph. Default units for indents *i* are ens.

Type font and size are reset to default values before each paragraph, and after processing font and size setting macros.

These strings are predefined by **−man**:

*R '®', '(Reg)' in *nroff.*

*S Change to default type size.

FILES

/usr/lib/tmac/tmac.an
/usr/man/man0/xx

SEE ALSO

troff(1), man(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).

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

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

Output of the *eqn, neqn, refer,* and *tbl*(1) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac/tmac.s

SEE ALSO

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

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® 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), termlib(3), ttytype(5)
troff(1) for *nroff*

BUGS

The programs that ought to adhere to this nomenclature do so only fitfully.

NAME

ac − login accounting

SYNOPSIS

/etc/ac [**−w** wtmp] [**−p**] [**−d**] [people] ...

DESCRIPTION

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

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

FILES

/usr/adm/wtmp

SEE ALSO

init(8), sa(8), login(1), utmp(5).

NAME

adduser — procedure for adding new users

DESCRIPTION

A new user must choose a login name, which must not already appear in */etc/passwd*. An account can be added by editing a line into the passwd file; this must be done with the password file locked e.g. by using *vipw*(8).

A new user is given a group and user id. User id's should be distinct across a system, since they are used to control access to files. Typically, users working on similar projects will be put in the same group. Thus at UCB we have groups for system staff, faculty, graduate students, and a few special groups for large projects. System staff is group "10" for historical reasons, and the super-user is in this group.

A skeletal account for a new user "ernie" would look like:

    ernie::235:20:& Kovacs,508E,7925,6428202:/mnt/grad/ernie:/bin/csh

The first field is the login name "ernie". The next field is the encrypted password which is not given and must be initialized using *passwd*(1). The next two fields are the user and group id's. Traditionally, users in group 20 are graduate students and have account names with numbers in the 200's. The next field gives information about ernie's real name, office and office phone and home phone. This information is used by the *finger*(1) program. From this information we can tell that ernie's real name is "Ernie Kovacs" (the & here serves to repeat "ernie" with appropriate capitalization), that his office is 508 Evans Hall, his extension is x2-7925, and this his home phone number is 642-8202. You can modify the *finger*(1) program if necessary to allow different information to be encoded in this field. The UCB version of finger knows several things particular to Berkeley — that phone extensions start "2−", that offices ending in "E" are in Evans Hall and that offices ending in "C" are in Cory Hall.

The final two fields give a login directory and a login shell name. Traditionally, user files live on a file system which has the machines single letter *net*(1) address as the first of two characters. Thus on the Berkeley CS Department VAX, whose Berknet address is "csvax" abbreviated "v" the user file systems are mounted on "/va", "/vb", etc. On each such filesystem there are subdirectories there for each group of users, i.e.: "/va/staff" and "/vb/prof". This is not strictly necessary but keeps the number of files in the top level directories reasonably small.

The login shell will default to "/bin/sh" if none is given. Most users at Berkeley choose "/bin/csh" so this is usually specified here.

It is useful to give new users some help in getting started, supplying them with a few skeletal files such as *.profile* if they use "/bin/sh", or *.cshrc* and *.login* if they use "/bin/csh". The directory "/usr/skel" contains skeletal definitions of such files. New users should be given copies of these files which, for instance, arrange to use *tset*(1) automatically at each login.

FILES

/etc/passwd	password file
/usr/skel	skeletal login directory

SEE ALSO

passwd(1), finger(1), chsh(1), chfn(1), passwd(5), vipw(8)

BUGS

User information should be stored in its own data base separate from the password file.

NAME

analyze — Virtual UNIX postmortem crash analyzer

SYNOPSIS

/etc/analyze [**−s** swapfile] [**−f**] [**−m**] [**−d**] [**−D**] [**−v**] corefile [system]

DESCRIPTION

Analyze is the post-mortem analyzer for the state of the paging system. In order to use *analyze* you must arrange to get a image of the memory (and possibly the paging area) of the system after it crashes (see *crash*(8)).

The *analyze* program reads the relevant system data structures from the core image file and indexing information from **/vmunix** (or the specified file). to determine the state of the paging subsystem at the point of crash. It looks at each process in the system, and the resources each is using in an attempt to determine inconsistencies in the paging system state. Normally, the output consists of a sequence of lines showing each active process, its state (whether swapped in or not), its *p0br,* and the number and location of its page table pages. Any pages which are locked while raw i/o is in progress, or which are locked because they are *intransit* are also printed. (Intransit text pages often diagnose as duplicated; you will have to weed these out by hand.)

The program checks that any pages in core which are marked as not modified are, in fact, identical to the swap space copies. It also checks for non-overlap of the swap space, and that the core map entries correspond to the page tables. The state of the free list is also checked.

Options to *analyze:*

−D causes the diskmap for each process to be printed.

−d causes the (sorted) paging area usage to be printed.

−f which causes the free list to be dumped.

−m causes the entire coremap state to be dumped.

−v (long unused) which causes a hugely verbose output format to be used.

In general, the output from this program can be confused by processes which were forking, swapping, or exiting or happened to be in unusual states when the crash occurred. You should examine the flags fields of relevant processes in the output of a *pstat*(8) to weed out such processes.

It is possible to look at the core dump with *adb* if you do

```
adb /vmunix /vmcore
/m 80000000 #ffffffff
```

which fixes the map of *vmcore* so that symbols in data space will work. Note that the debugger is looking at the physical memory at the point of crash; you will have to determine which pages of physical memory virtual pages are in if you wish to look at them. If *analyze* says that a processes page tables are in page 218 (hex of course), then you can look at them by looking at address 0x80043000 in the dump, i.e. "80043000,80/X" will print the page of page tables.

FILES

/vmunix default system namelist

SEE ALSO

ps(1), crash(8), pstat(8)

AUTHORS

Ozalp Babaoglu and William Joy

DIAGNOSTICS

Various diagnostics about overlaps in swap mappings, missing swap mappings, page table entries inconsistent with the core map, incore pages which are marked clean but differ from disk-image copies, pages which are locked or intransit, and inconsistencies in the free list.

It would be nice if this program analyzed the system in general, rather than just the paging system in particular.

NAME

arcv — convert archives to new format

SYNOPSIS

/etc/arcv file ...

DESCRIPTION

Arcv converts archive files (see *ar*(1), *ar*(5)) from 32v and Third Berkeley editions to a new portable format. The conversion is done in place, and the command refuses to alter a file not in old archive format.

Old archives are marked with a magic number of 0177545 at the start; new archives have a first line "!<arch>".

FILES

/tmp/v*, temporary copy

SEE ALSO

ar(1), ar(5)

NAME

arff, flcopy − archiver and copier for floppy

SYNOPSIS

/etc/arff [key] [name ...]
/etc/flcopy [**−h**] [**−t***n*]

DESCRIPTION

Arff saves and restores files on the console floppy disk. 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 names specifying which files are to be dumped or restored.

Files names have restrictions, because of radix50 considerations. They must be in the form 1-6 alphanumerics followed by "." followed by 0-3 alphanumerics. Case distinctions are lost. Only the trailing component of a pathname is used.

The function portion of the key is specified by one of the following letters:

r The named files are replaced where found on the floppy, or added taking up the minimal possible portion of the first empty spot on the floppy.

x The named files are extracted from the floppy.

d The named files are deleted from the floppy. Arff will combine contiguous deleted files into one empty entry in the rt-11 directory.

t The names of the specified files are listed each time they occur on the floppy. If no file argument is given, all of the names on the floppy are listed.

The following characters may be used in addition to the letter which selects the function desired.

v Normally *arff* 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 floppy entries than just the name.

f causes *arff* to use the next argument as the name of the archive instead of /dev/floppy.

m causes *arff* not to use the mapping algorithm employed in interleaving sectors around a floppy disk. In conjunction with the **f** option it may be used for extracting files from rt11 formatted cartridge disks, for example.

Flcopy copies the console floppy disk (opened as '/dev/floppy') to a file created in the current directory, named "floppy", then prints the message "Change Floppy, hit return when done". Then *flcopy* copies the local file back out to the floppy disk.

The **−h** option to *flcopy* causes it to open a file named "floppy" in the current directory and copy it to */dev/floppy;* the **−t** option causes only the first *n* tracks to participate in a copy.

FILES

/dev/floppy
floppy (in current directory)

AUTHORS

Keith Sklower, Richard Tuck

BUGS

Floppy errors are handled ungracefully; *Arff* does not handle multi-segment rt11 directories.

NAME

bad144 — read/write dec standard 144 bad sector information

SYNOPSIS

bad144 disktype disk [sno [bad ...]]

DESCRIPTION

Bad144 can be used to inspect the information stored on a disk that is used by the disk drivers to implement bad sector forwarding. The format of the information is specified by DEC standard 144, as follows.

The bad sector information is located in the first 5 even numbered sectors of the last track of the disk pack. There are five identical copies of the information, described by the dkbad structure.

Replacement sectors are allocated starting with the first sector before the bad sector information and working backwards towards the beginning of the disk. A maximum of 126 bad sectors are supported. The position of the bad sector in the bad sector table determines which replacement sector it corresponds to.

The bad sector information and replacement sectors are conventionally only accessable through the "c" file system partition of the disk. If that partition is used for a file system, the user is responsible for making sure that it does not overlap the bad sector information or any replacement sectors.

The bad sector structure is as follows:

```
struct dkbad {
        long        bt_csn;                    /* cartridge serial number */
        u_short     bt_mbz;                    /* unused; should be 0 */
        u_short     bt_flag;                   /* -1 => alignment cartridge */
        struct bt_bad {
                u_short bt_cyl;                /* cylinder number of bad sector */
                u_short bt_trksec;             /* track and sector number */
        } bt_bad[126];
};
```

Unused slots in the bt_bad array are filled with all bits set, a putatively illegal value.

Bad144 is invoked by giving a device type (e.g. rk07, rm03, rm05, etc.), and a device name (e.g. hk0, hp1, etc.). It reads the first sector of the last track of the corresponding disk and prints out the bad sector information. It may also be invoked giving a serial number for the pack and a list of bad sectors, and will then write the supplied information onto the same location. Note, however, that *bad144* does not arrange for the specified sectors to be marked bad in this case. This option should only be used to restore known bad sector information which was destroyed.

New bad sectors can be added by running the standard DEC formatter in section "bad".

SEE ALSO

badsect (8), format (8)

BUGS

It should be possible to both format disks on-line under UNIX and to change the bad sector information, marking new bad sectors, without running a standalone program.

The bootstrap drivers used to boot the system do not understand bad sectors, handle ECC errors, or the special SSE (skip sector) errors of RM80 type disks. This means that none of these errors can occur when reading the file /vmunix to boot. When a disk drive is used to load the bootstrap code (the alternative would be that the bootstrap would be loaded from the console media), sector 0 of the disk drive and the file /boot in the root file system of that drive must also not have any of these errors in it.

The drivers which write a system core image on disk after a crash do not handle errors; thus the crash dump area must be free of errors and bad sectors.

NAME

badsect — create files to contain bad sectors

SYNOPSIS

/etc/badsect sector ...

DESCRIPTION

Badsect makes a file to contain a bad sector. Normally, bad sectors are made inaccessible by the standard formatter, which provides a forwarding table for bad sectors to the driver; see *bad144*(8) for details. If a driver supports the bad blocking standard it is much preferable to use that method to isolate bad blocks, since the bad block forwarding makes the pack appear perfect, and such packs can then be copied with *dd*(8). The technique used by this program is also less general than bad block forwarding, as *badsect* can't make amends for bad blocks in the i-list of file systems or in swap areas.

Adding a sector which is suddenly bad to the bad sector table currently requires the running of the standard DEC formatter, as UNIX does not supply formatters. Thus to deal with a newly bad block or on disks where the drivers do not support the bad-blocking standard *badsect* may be used to good effect.

Badsect is used on a quiet file system in the following way: First mount the file system, and change to its root directory. Make a directory BAD there and change into it. Run badsect giving as argument all the bad sectors you wish to add. (The sector numbers should be relative to the beginning of the file system, but this is not hard to do as the system reports relative sector numbers in its console error messages.) Then change back to the root directory, unmount the file system and run *fsck*(8) on the file system. The bad sectors should show up in two files or in the bad sector files and the free list. Have *fsck* remove files containing the offending bad sectors, but **do not** have it remove the BAD/*nnnnn* files. This will leave the bad sectors in only the BAD files.

Badsect works by giving the specified sector numbers in a *mknod*(2) system call, creating a regular file whose first block address is the block containing bad sector and whose name is the bad sector number. The file has 0 length, but the check programs will still consider it to contain the block containing the sector. This has the pleasant effect that the sector is completely inaccessible to the containing file system since it is not available by accessing the file.

SEE ALSO

bad144(8), fsck (8)

BUGS

If both sectors which comprise a (1024 byte) disk block are bad, you should specify only one of them to *badsect*, as the blocks in the bad sector files actually cover both (bad) disk sectors.

NAME

catman — create the cat files for the manual

SYNOPSIS

/etc/catman [**−p**] [**−n**] [**−w**] [sections]

DESCRIPTION

Catman creates the preformatted versions of the on-line manual from the nroff input files. Each manual page is examined and those whose preformatted versions are missing or out of date are recreated. If any changes are made, *catman* will recreate the **/usr/lib/whatis** database.

If there is one parameter not starting with a '−', it is take to be a list of manual sections to look in. For example

catman 123

will cause the updating to only happen to manual sections 1, 2, and 3.

Options:

- **−n** prevents creations of **/usr/lib/whatis**.
- **−p** prints what would be done instead of doing it.
- **−w** causes only the **/usr/lib/whatis** database to be created. No manual reformatting is done.

FILES

/usr/man/man?/*.*	raw (nroff input) manual sections
/usr/man/cat?/*.*	preformatted manual pages
/usr/lib/makewhatis	commands to make whatis database

SEE ALSO

man(1)

BUGS

Acts oddly on nights with full moons.

NAME

chown, chgrp — change owner or group

SYNOPSIS

/etc/chown owner file ...

/etc/chgrp group file ...

DESCRIPTION

Chown changes the owner of the *files* to *owner*. The owner may be either a decimal UID or a login name found in the password file.

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 owner or group, in order to simplify as yet unimplemented accounting procedures.

FILES

/etc/passwd
/etc/group

SEE ALSO

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

NAME

clri — clear i-node

SYNOPSIS

/etc/clri filesystem i-number ...

DESCRIPTION

N.B.: *Clri* is obsoleted for normal file system repair work by *fsck*(8).

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

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

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

SEE ALSO

icheck(8)

BUGS

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

NAME

config — Build system configuration files

SYNOPSIS

/etc/config *config_file*

DESCRIPTION

Config builds a set of system configuration files from a short file which describes the sort of system that is being configured. It also takes as input a file which tells *config* what files are needed to generate a system.

Config should be run from the **conf** subdirectory of the system source (usually /sys/conf or /usr/src/sys/conf). *Config* assumes that there is already a directory *../config_file* created and it places all its output files in there. The output of *config* consists of a number files: **ioconf.c** which contains a description of what i/o devices are attached to the system, **ubglue.s** which is a set of interrupt service routines for devices attached to the UNIBUS, **makefile** for building the system and a set of header files which contain the number of various devices that will be compiled into the system.

After running *config,* it is necessary to run "make depend" in the directory where the new makefile was created. *Config* reminds you of this when it completes.

If you get any other error messages from *config,* you should fix the problems in your configuration file and try again. If you try to compile a system that had configuration errors, you will meet with failure.

CONFIG FILE FORMAT

In the following descriptions, a number can be a decimal integer, a whole octal number or a whole hexadecimal number. Hex and octal are specified to *config* in the same way they are specified to the C compiler, a number starting with "0x" is a hex number and a number starting with just a "0" is an octal number. When specifying the timezone, you may also use floating point numbers.

Comments are specified in a config file with the character "#". All characters from a "#" to the end of a line are ignored.

Lines beginning with tabs are considered continuations of the previous line.

Lines of the config file can be one of several types. First there are lines which describe general things about your system. Here is a list of the possibilities.

cpu "*type*"
: This system is to run on the cpu type specified. More than one cpu type can appear in the config file. Legal types are **VAX780, VAX750,** and **VAX7ZZ.**

options *optlist*
: Compile the listed options into the system. Options in this list are seperated by commas. There is a list of options that you may specify in the generic makefile. A line of the form "options FUNNY,HAHA" yields −DFUNNY −DHAHA to the C compiler.

timezone *number* [**dst**]
: Specifies the timezone you are in. This is measured in the number of hours west of GMT you are. 5 is EST, 8 is PST. If you specify **dst,** the system will operate under daylight savings time.

ident *name*
: This system is to be known as *name.* This is usually a cute name like ERNIE (short for Ernie Co-Vax) or VAXWELL (for Vaxwell Smart).

maxusers *number*
The maximum expected number of simultaneously active user on this system is *number.* This number is used to size several system data structures.

config *device sysname*
Generate a system which runs with its root on *device* and call it *sysname.* There may be more than one *config* specification in a config file.

The second type of line in the config file describes what devices your system has and what they are connected to (e.g. I have a DZ-11 on UNIBUS Adapter 0). These lines have the following format.

dev_type dev_name **at** *con_dev more_info*

Dev_type is either **master, tape, disk, controller device,** or **pseudo-device.** A master is a MASSBUS tape controller. A controller is a disk controller, a UNIBUS tape controller, an mba (MASSBUS) or a uba (UNIBUS). A device is usually something which connects to the uba, like a DZ-11 or a DR-11. Disk and tape should be self-explanatory. A pseudo-device is something which should be conditionally loaded, but is not really a device. Current examples are the bk line discipline, the pseudo-tty driver and various network subsystems. If you load a subsystem you will probably find it convenient to enable conditionaly code using a **options** specification.

The *dev_name* is the name of the device you are specifying. If it is not a pseudo-device, you must give a number afterwards (e.g. dz0, dz1, hp0).

Con_dev is what the device you are specifying is connected to. If you have a disk on MASSBUS adapter zero then the proper *con_dev* is **mba0.** For MASSBUS and UNIBUS adapters, you must give **nexus?** as the *con_dev.*

The *more_info* field is a sequence of the following:

csr *addr*
Specifies the csr for a device. Must be given for UNIBUS tape and disk controllers and all devices connected to the UNIBUS. Make certain that you put a leading zero on the address so that it will be interpreted as an octal number.

drive *number*
For a disk or UNIBUS tape, specifies which drive this is.

slave *number*
For a MASSBUS tape, specifies which tape slave it is.

flags *number*
These flags are passed to the device driver at system initialization time.

vector *addr [addr]*
For devices which interrupt on the UNIBUS, specifies the interrupt service routine.

The easiest way to understand config files it to look at a working one and modify it to suit your system. Here is a short sample configuration file for a system with an RM03, a TU45, a DZ-11 and a DH-11.

```
#
# Sample configuration file
#

cpu         VAX780
ident       SAMPLE
hz          60
timezone    8 dst
maxusers    24
```

```
config          hp      vmunix
config          rk      rkvmunix

controller      mba0    at nexus ?
controller      uba0    at nexus ?
disk            hp0     at mba0 drive 0
master          ht0     at mba1 drive 0
tape            tu0     at ht0 slave 0
pseudo-device   pty
pseudo-device   bk
controller      hk0     at uba0 csr 0177440               vector rkintr
disk            rk0     at hk0 drive 0
disk            rk1     at hk0 drive 1
device          dh1     at uba0 csr 0160040               vector dhrint dhxint
device          dz0     at uba0 csr 0160100 flags 0xc0    vector dzrint dzxint
```

A ? may be substituted for a number in two places and the system will figure out what to fill in for the ? when it boots. You can put question marks on a *con_dev* (e.g. at mba?) or on a drive number (e.g. drive ?). This allows redundancy as a single system can be built which will reboot on different hardware configurations.

FILES

/sys/conf/makefile   Generic makefile
/sys/conf/files   List of files system is built from

SEE ALSO

The SYNOPSIS portion of each device in section 4.

AUTHOR

Michael Toy

BUGS

The line numbers reported in error messages are usually off by one.

Should describe the format of the "files" file here; you can probably figure it out for yourself in the meantime.

No exhaustive testing of responses to all the weird input semantic errors possible has been done.

NAME

crash − what happens when the system crashes

DESCRIPTION

This section explains what happens when the system crashes and how you can analyze crash dumps.

When the system crashes voluntarily it prints a message of the form

panic: why i gave up the ghost

on the console, takes a dump on a mass storage peripheral, and then invokes an automatic reboot procedure as described in *reboot*(8). (If auto-reboot is disabled on the front panel of the machine the system will simply halt at this point.) Unless some unexpected inconsistency is encountered in the state of the file systems due to hardware or software failure the system will then resume multi-user operations.

The system has a large number of internal consistency checks; if one of these fails, then it will panic with a very short message indicating which one failed.

The most common cause of system failures is hardware failure, which can reflect itself in different ways. Here are the messages which you are likely to encounter, with some hints as to causes. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

IO err in push
hard IO err in swap

The system encountered an error trying to write to the paging device or an error in reading critical information from a disk drive. You should fix your disk if it is broken or unreliable.

timeout table overflow

This really shouldn't be a panic, but until we fix up the data structure involved, running out of entries causes a crash. If this happens, you should make the timeout table bigger.

KSP not valid
SBI fault
CHM? in kernel

These indicate either a serious bug in the system or, more often, a glitch or failing hardware. If SBI faults recur, check out the hardware or call field service. If the other faults recur, there is likely a bug somewhere in the system, although these can be caused by a flakey processor. Run processor microdiagnostics.

machine check %x: *description*
machine dependent machine-check information

We should describe machine checks, and will someday. For now, ask someone who knows (like your friendly field service people).

trap type %d, code=%d, pc=%x

A unexpected trap has occurred within the system; the trap types are:

0 reserved addressing fault
1 privileged instruction fault
2 reserved operand fault
3 bpt instruction fault
4 xfc instruction fault
5 system call trap
6 arithmetic trap
7 ast delivery trap

8	segmentation fault
9	protection fault
10	trace trap
11	compatibility mode fault
12	page fault
13	page table fault

The favorite trap type in system crashes is trap type 9, indicating a wild reference. The code is the referenced address, and the pc at the time of the fault is printed. These problems tend to be easy to track down if they are kernel bugs since the processor stops cold, but random flakiness seems to cause this sometimes, e.g. we have trapped with code 80000800 three times in six months as an instruction fetch went across this page boundary in the kernel but have been unable to find any reason for this to have happened.

init died

The system initialization process has exited. This is bad news, as no new users will then be able to log in. Rebooting is the only fix, so the system just does it right away.

That completes the list of panic types you are likely to see.

When the system crashes it write (or at least attempts to write) a image of the current memory into the back end of the primary swap area. After the system is rebooted, the program *savecore*(8) runs and preserves a copy of this core image and the current system in a specified directory for later perusal. See *savecore*(8) for details.

To analyze a dump you should begin by running *ps −alxk* to print the process table at the time of the crash. Use *adb*(1) to examine */vmcore*. The location *_rpb+0t508* is the bottom of a stack onto which were pushed the stack pointer **sp**, **PCBB** (containing the physical address of a *u_area*), **MAPEN**, **IPL**, and registers **r13−r0** (in that order). **r13**(fp) is the system frame pointer and the stack is used in standard **calls** format. Use *adb*(1) to get a reverse calling order. In most cases this procedure will give an idea of what is wrong. A more complete discussion of system debugging is impossible here. See, however, *analyze*(8) for some more hints.

SEE ALSO

analyze(8), reboot(8)
VAX 11/780 System Maintenance Guide for more information about machine checks.

BUGS

There should be a better program than *analyze*(8) available which prints out more of the system state symbolically after a crash to lessen the tedious tasks involved in crash analysis.

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*(8).

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

Crontab is examined by *cron* every minute.

FILES

/usr/lib/crontab

NAME

dcheck — file system directory consistency check

SYNOPSIS

/etc/dcheck [**−i** numbers] [filesystem]

DESCRIPTION

N.B.: *Dcheck* is obsoleted for normal consistency checking by *fsck*(8).

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

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

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

FILES

Default file systems vary with installation.

SEE ALSO

fsck(8), icheck(8), filsys(5), clri(8), ncheck(8)

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 is obsoleted by *fsck* and remains for historical reasons.

NAME

delivermail — deliver mail to arbitrary people

SYNOPSIS

/etc/delivermail [**—[fr]** *address*] [**—a**] [**—e***x*] [**—n**] [**—m**] [**—s**] [**—i**] [**—h** *N*] address ...

DESCRIPTION

Delivermail delivers a letter to one or more people, routing the letter over whatever networks are necessary. *Delivermail* will do inter-net forwarding as necessary to deliver the mail to the correct place.

Delivermail is not intended as a user interface routine; it is expected that other programs will provide user-friendly front ends, and *delivermail* will be used only to deliver pre-formatted messages.

Delivermail reads its standard input up to a control-D or a line with a single dot and sends a copy of the letter found there to all of the addresses listed. If the **—i** flag is given, single dots are ignored. It determines the network to use based on the syntax of the addresses. Addresses containing the character '@' or the word "at" are sent to the ARPANET; addresses containing '!' are sent to the UUCP net, and addresses containing ':' or '.' are sent to the Berkeley network. Other addresses are assumed to be local.

Local addresses are looked up in a file constructed by *newaliases*(1) from the data file */usr/lib/aliases* and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash or using the **—n** flag. Normally the sender is not included in any alias expansions, e.g., if 'john' sends to 'group', and 'group' includes 'john' in the expansion, then the letter will not be delivered to 'john'. The **—m** flag disables this suppression.

Delivermail computes the person sending the mail by looking at your login name. The "from" person can be explicitly specified by using the **—f** flag; or, if the **—a** flag is given, delivermail looks in the body of the message for a "From:" or "Sender:" field in ARPANET format. The **—f** and **—a** flags can be used only by the special users *root* and *network*, or if the person you are trying to become is the same as the person you are. The **—r** flag is entirely equivalent to the **—f** flag; it is provided for ease of interface only.

The **—e***x* flag controls the disposition of error output, as follows:

e Print errors on the standard output, and echo a copy of the message when done. It is assumed that a network server will return the message back to the user.

m Mail errors back to the user.

p Print errors on the standard output.

q Throw errors away; only exit status is returned.

w Write errors back to the user's terminal, but only if the user is still logged in and write permission is enabled; otherwise errors are mailed back.

If the error is not mailed back, and if the mail originated on the machine where the error occurred, the letter is appended to the file *dead.letter* in the sender's home directory.

If the first character of the user name is a vertical bar, the rest of the user name is used as the name of a program to pipe the mail to. It may be necessary to quote the name of the user to keep *delivermail* from suppressing the blanks from between arguments.

The message is normally edited to eliminate "From" lines that might confuse other mailers. In particular, "From" lines in the header are deleted, and "From" lines in the body are prepended by '>'. The **—s** flag saves "From" lines in the header.

The **−h** flag gives a "hop-count", i.e., a measure of how many times this message has been processed by *delivermail* (presumably on different machines). Each time *delivermail* processes a message, it increases the hop-count by one; if it exceeds 30 *delivermail* assumes that an alias loop has occurred and it aborts the message. The hop-count defaults to zero.

Delivermail returns an exit status describing what it did. The codes are defined in <sysexits.h>

EX_OK	Successful completion on all addresses.
EX_NOUSER	User name not recognized.
EX_UNAVAILABLE	Catchall meaning necessary resources were not available.
EX_SYNTAX	Syntax error in address.
EX_SOFTWARE	Internal software error, including bad arguments.
EX_OSERR	Temporary operating system error, such as "cannot fork".
EX_NOHOST	Host name not recognized.

FILES

/usr/lib/aliases	raw data for alias names
/usr/lib/aliases.dir	data base of alias names
/usr/lib/aliases.pag	
/bin/mail	to deliver uucp mail
/usr/net/bin/v6mail	to deliver local mail
/usr/net/bin/sendmail	to deliver Berknet mail
/usr/lib/mailers/arpa	to deliver ARPANET mail
/tmp/mail*	temp file
/tmp/xscript*	saved transcript

SEE ALSO

biff(1), binmail(1), mail(1), newaliases(1), aliases(5)

BUGS

Delivermail sends one copy of the letter to each user; it should send one copy of the letter to each host and distribute to multiple users there whenever possible.

Delivermail assumes the addresses can be represented as one word. This is incorrect according to the ARPANET mail protocol RFC 733 (NIC 41952), but is consistent with the real world.

NAME

dmesg – collect system diagnostic messages to form error log

SYNOPSIS

/etc/dmesg [–]

DESCRIPTION

Dmesg looks in a system buffer for recently printed diagnostic messages and prints them on the standard output. The messages are those printed by the system when device (hardware) errors occur and (occasionally) when system tables overflow non-fatally. If the – flag is given, then *dmesg* computes (incrementally) the new messages since the last time it was run and places these on the standard output. This is typically used with *cron*(8) to produce the error log */usr/adm/messages* by running the command

```
/etc/dmesg – >> /usr/adm/messages
```

every 10 minutes.

FILES

/usr/adm/messages	error log (conventional location)
/usr/adm/msgbuf	scratch file for memory of – option

BUGS

The system error message buffer is of small finite size. As *dmesg* is run only every few minutes, not all error messages are guaranteed to be logged. This can be construed as a blessing rather than a curse.

Error diagnostics generated immediately before a system crash will never get logged.

NAME

dump – incremental file system dump

SYNOPSIS

/etc/dump [key [*argument* ...] filesystem]

DESCRIPTION

Dump copies to magnetic tape all files changed after a certain date in the *filesystem*. The *key* specifies the date and other options about the dump. *Key* consists of characters from the set **0123456789fuJsdWn.**

0–9 This number is the 'dump level'. All files modified since the last date stored in the file */etc/dumpdates* 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 instead of the tape.

u If the dump completes successfully, write the date of the beginning of the dump on file */etc/dumpdates*. This file records a separate date for each filesystem and each dump level. The format of */etc/dumpdates* is readable by people, consisting of one free format record per line: filesystem name, increment level and *ctime(3)* format dump date. */etc/dumpdates* may be edited to change any of the fields, if necessary. Note that */etc/dumpdates* is in a format different from that which previous versions of *dump* maintained in */etc/ddate*, although the information content is identical.

J This option is intended to be invoked only when the old format */etc/ddate* files are updated to the new format */etc/dumpdates* format. The effect of this option is to convert between the old, obsolete format and to the new format. If the **J** option is invoked, all other options are ignored, and *dump* terminates immediately.

s The size of the dump tape is specified in feet. The number of feet is taken from the next *argument*. When the specified size is reached, *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.

W *Dump* tells the operator what file systems need to be dumped. This information is gleaned from the files */etc/dumpdates* and */etc/fstab*. The **W** option causes *dump* to print out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped. If the **W** option is set, all other options are ignored, and *dump* exits immediately.

w Is like W, but prints only those filesystems which need to be dumped.

n Whenever *dump* requires operator attention, notify by means similar to a *wall(1)* all of the operators in the group "operator".

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 tape, end of dump, tape write error, tape open error or disk read error (if there are more than a threshold of 32). In addition to alerting all operators implied by the **n** key, *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.

Since making a dump involves a lot of time and effort for full dumps, *dump* checkpoints itself at the start of each tape volume. If writing that volume fails for some reason, *dump* will, with operator permission, restart itself from the checkpoint after the old tape has been rewound and removed, and a new tape has been mounted.

Dump tells the operator what is going on at periodic intervals, including usually low estimates of the number of blocks to write, the number of tapes it will take, the time to completion, and the time to the tape change. The output is verbose, so that others know that the terminal controlling *dump* is busy, and will be for some time.

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

    dump 0un

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 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 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 tapes that is saved forever.

FILES

/dev/rrp1g	default filesystem to dump from
/dev/rmt8	default tape unit to dump to
/etc/ddate	old format dump date record (obsolete after **−J** option)
/etc/dumpdates	new format dump date record
/etc/fstab	Dump table: file systems and frequency
/etc/group	to find group *operator*

SEE ALSO

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

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. Fewer than 32 read errors on the filesystem are ignored. Each reel requires a new process, so parent processes for reels already written just hang around until the entire tape is written.

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

SYNOPSIS

/etc/dumpdir [**f** filename]

DESCRIPTION

Dumpdir is used to read magtapes dumped with the *dump* command and list the names and inode numbers of all the files and directories on the tape.

The **f** option causes *filename* as the name of the tape instead of the default.

FILES

default tape unit varies with installation
rst*

SEE ALSO

dump(1), restor(1)

DIAGNOSTICS

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

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *dumpdir* doesn't use it.

NAME

format − how to format disks

DESCRIPTION

Warning: These instructions are for people with 11/780 CPU's. We don't know how to do this for 11/750 cpus (yet at least), you'll have to figure it out yourself; if you do call us and tell us how.

The formatting procedures are different for each type of disk. Listed here are the formatting procedures for RK07's, RP0X, RM0X and Emulex Unibus Disks.

You should shut down UNIX and halt the machine to do any disk formatting. Make certain you put in the pack you want formatted. It is also a good idea to spin down or write protect the disks you don't want to format, just in case.

Formatting a RK07. Load the floppy labled, "RX11 VAX DSK LD DEV #1" in the floppy disk drive, and type the following commands:

```
>>>BOOT
DIAGNOSTIC SUPERVISOR.  ZZ-ESSAA-X5.0-119  23-JAN-1980 12:44:40.03
DS>ATTACH DW780 SBI DW0 3 5
DS>ATTACH RK07 DW0 DMA0
DS>SELECT DMA0
DS>LOAD EVRAC
DS>START/SEC:PACKINIT
```

Formatting a RP0X. Follow the above procedures except that the ATTACH and SELECT lines should read.

```
DS>ATTACH RH780 SBI RH0 8 5
DS>ATTACH RP0X RH0 DBA0              (RP0X is, e.g. RP06)
DS>SELECT DBA0
```

This is for drive 0 on mba0; use 9 instead of 8 for mba1, etc.

Formatting a RM0X. Follow the above procedures except that the ATTACH and SELECT lines should read.

```
DS>ATTACH RH780 SBI RH0 8 5
DS>ATTACH RM0X RH0 DRA0
DS>SELECT DRA0
```

Formatting an Emulex Unibus Disk. Type these commands on the console:

```
>>>SET REL:2013FDC0
>>>SET DEF WORD
>>>SET DEF OCT
>>>SET DEF PHYS
>>>U
>>>I
>>>D/P 10 0
>>>D/P 0 21
>>>D/P 36 177777
>>>D/P 0 77
(figure out when it is done)
>>>SET REL:0
>>>SET DEF LONG
>>>SET DEF HEX
```

Once a disk is formatted, you'll still have to build file systems on it with mkfs(8) before you can use it with UNIX.

Don't forget to put your UNIX console floppy back in the floppy disk drive.

SEE ALSO

bad144(8), badsect(8), mkfs(8)

NAME

fsck − file system consistency check and interactive repair

SYNOPSIS

/etc/fsck −p [filesystem ...]
/etc/fsck [**−y**] [**−n**] [**−s**X] [−SX] [**−t** filename] [filesystem] ...

DESCRIPTION

The first form of *fsck* preens a standard set of filesystems or the specified file systems. It is normally used in the script /etc/rc during automatic reboot. In this case *fsck* reads the table **/etc/fstab** to determine which file systems to check. It uses the information there to inspect groups of disks in parallel taking maximum advantage of i/o overlap to check the file systems as quickly as possible. Normally, the root file system will be checked on pass 1, other "root" ("a" partition) file systems on pass 2, other small file systems on separate passes (e.g. the "d" file systems on pass 3 and the "e" file systems on pass 4), and finally the large user file systems on the last pass, e.g. pass 5. A pass number of 0 in fstab causes a disk to not be checked; similarly partitions which are not shown as to be mounted "rw" or "ro" are not checked.

The system takes care that only a restricted class of innocuous inconsistencies can happen unless hardware or software failures intervene. These are limited to the following:

Unreferenced inodes

Link counts in inodes too large

Missing blocks in the free list

Blocks in the free list also in files

Counts in the super-block wrong

These are the only inconsistencies which *fsck* with the **−p** option will correct; if it encounters other inconsistencies, it exits with an abnormal return status and an automatic reboot will then fail. For each corrected inconsistency one or more lines will be printed identifying the file system on which the correction will take place, and the nature of the correction. After successfully correcting a file system, *fsck* will print the number of files on that file system and the number of used and free blocks.

Without the **−p** option, *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 which are not fixable under the **−p** option 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/fstab**.

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.

FILES

/etc/fstab contains default list of file systems to check.

DIAGNOSTICS

The diagnostics produced by *fsck* are intended to be self-explanatory.

SEE ALSO

fstab(5), fs(5), crash(8), reboot(8)

BUGS

Inode numbers for . and .. in each directory should be checked for validity.

−g and **−b** options from *check* should be available in *fsck*.

There should be some way to start a **fsck −p** at pass *n*.

NAME

getty — set terminal mode

SYNOPSIS

/etc/getty [char]

DESCRIPTION

Getty is invoked by *init*(8) 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. Arguments other than '0' can be used to make *getty* treat the line specially. Normally, it sets the speed of the interface to 300 baud, specifies that raw mode is to be used (break on every character), that echo is to be suppressed, and either parity allowed. It types a banner identifying the system (from /usr/include/ident.h and the 'login:' message. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the 'break' ('interrupt') key. The speed is then changed to 1200 baud and the 'login:' is typed again; a second 'break' changes the speed to 150 baud and the 'login:' is typed again. Successive 'break' characters cycle through the speeds 300, 1200, and 150 baud.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *stty*(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(8), login(1), stty(2), ttys(5)

BUGS

NAME

halt – stop the processor

SYNOPSIS

/etc/halt [**–n**] [**–q**] [**–y**]

DESCRIPTION

Halt writes out sandbagged information to the disks and then stops the processor. The machine does not reboot, even if the auto-reboot switch is set on the console.

The **–n** option prevents the sync before stopping. The **–q** option causes a quick halt, no graceful shutdown is attempted. The **–y** option is needed if you are trying to halt the system from a dialup.

SEE ALSO

reboot(8), shutdown(8)

BUGS

It is very difficult to halt a VAX, as the machine wants to then reboot itself. A rather tight loop suffices.

NAME

icheck − file system storage consistency check

SYNOPSIS

/etc/icheck [**−s**] [**−b** numbers] [filesystem]

DESCRIPTION

N.B.: *Icheck* is obsoleted for normal consistency checking by *fsck*(8).

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

> The total number of files and the numbers of regular, directory, block special and character special files.
>
> The total number of blocks in use and the numbers of single-, double-, and triple-indirect blocks and directory blocks.
>
> The number of free blocks.
>
> The number of blocks missing; i.e. not in any file nor in the free list.

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

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

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

FILES

Default file systems vary with installation.

SEE ALSO

fsck(8), dcheck(8), ncheck(8), filsys(5), clri(8)

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

/etc/init

DESCRIPTION

Init is invoked inside UNIX as the last step in the boot procedure. It normally then runs the automatic reboot sequence as described in *reboot*(8), and if this succeeds, begins multi-user operation. If the reboot fails, it commences single user operation by giving the super-user a shell on the console. It is possible to pass parameters from the boot program to *init* so that single user operation is commenced immediately. When such single user operation is terminated by killing the single-user shell (i.e. by hitting ^D), *init* runs */etc/rc* without the reboot parameter. This command file performs housekeeping operations such as removing temporary files, mounting file systems, and starting daemons.

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. Opening the terminal will usually involve a delay, since the *open* is not completed until someone is dialed up and carrier established on the channel. If a terminal exists but an error occurs when trying to open the terminal *init* complains by writing a message to the system console; the message is repeated every 10 minutes for each such terminal until the terminal is shut off in /etc/ttys and init notified (by a hangup, as described below), or the terminal becomes accessible (init checks again every minute). 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.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. The *wtmp* entry is made only if a user logged in successfully on the line. 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 phone lines without rebooting the system by changing the *ttys* file and sending a *hangup* signal to the *init* process: use '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.

Init will cease creating new *getty*'s and allow the system to slowly die away, if it is sent a terminal stop (TSTP) signal, i.e. "kill −TSTP 1". A later hangup will resume full multi-user operations, or a terminate will initiate a single user shell. This hook is used by *reboot*(8) and *halt*(8).

Init's role is so critical that if it dies, the system will reboot itself automatically. If, at bootstrap time, the *init* process cannot be located, the system will loop in user mode at location 0x13.

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 (wont 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

SEE ALSO

login(1), kill(1), sh(1), ttys(5), crash(8), getty(8), rc(8), reboot(8), halt(8), shutdown(8)

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 '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

Makekey is intended for programs that perform encryption (e.g. *ed* and *crypt*(1)). Usually its input and output will be pipes.

SEE ALSO

crypt(1), ed(1)

NAME

mkfs − construct a file system

SYNOPSIS

/etc/mkfs special size [m n]
/etc/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. (No boot program is initialized in this form of *mkfs.)*

N.B.: All filesystems should have a *lost+found* directory for *fsck*(8); this should be created for each file system by running *mklost+found*(8) 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
        $
$
```

The arguments m and n specify the interleave factor. M should always be 3 and you should use the following table to choose n. as follows.

RM03	80
RM05	304
RM80	217
RP06	209
RP07	800
SI/CDC 9766	304
RK07	33
EMULEX/AMPEX 300M	304
EMULEX/FUJITSU 160M	160

SEE ALSO

filsys(5), dir(5), fsck(8), mklost+found(8)

BUGS

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
/etc/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*(8). This command should be run immediately after first mounting a newly created file system.

SEE ALSO
fsck(8), mkfs(8)

BUGS
Should be done automatically by *mkfs*.

NAME

mknod − build special file

SYNOPSIS

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

DESCRIPTION

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

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

SEE ALSO

mknod(2)

NAME

mount, umount — mount and dismount file system

SYNOPSIS

/etc/mount [special name [**−r**]]

/etc/mount −a

/etc/umount special

/etc/umount −a

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.

If the **−a** option is present for either *mount* or *umount,* all of the file systems described in */etc/fstab* are attempted to be mounted or unmounted. In this case, *special* and *name* are taken from */etc/fstab.* The *special* file name from */etc/fstab* is the block special name.

These commands maintain a table of mounted devices in */etc/mtab.* 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
/etc/fstab	file system table

SEE ALSO

mount(2), mtab(5), fstab(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

ncheck — generate names from i-numbers

SYNOPSIS

/etc/ncheck [**−i** numbers] [**−a**] [**−s**] [filesystem]

DESCRIPTION

N.B.: For most normal file system maintenance, the function of *ncheck* is subsumed by *fsck*(8).

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. suppressed. The **−s** option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

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

SEE ALSO

sort(1), dcheck(8), fsck(8), icheck(8)

DIAGNOSTICS

When the filesystem structure is improper, '??' denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

NAME

old − directory of old programs

SYNOPSIS

/usr/old/bin
/usr/old/include
/usr/old/lib

/usr/old/cc −I/usr/old/include ...

DESCRIPTION

After the 3rd Berkeley Distribution, the formats for binary and archive files were changed. The binaries were modified to allow arbitrary length symbols, which required adding a string table at the end of the symbol table, and having symbol table entries point into the names in that table. The archive was modified to be a portable format, using strings instead of binary numbers, to avoid problems of different sizes of integers on different machines. These changes are incompatible with older formats.

/usr/old is the root of a hierarchy of binaries, include files, and libraries in the old binary and archive formats. They contain a complete set of programs and files necessary for people who need to deal with the original UNIX formats.

In order to create new binaries in the old format, one must include the right header files. For example, to create a program called "foo" which uses the old math library in the old format, say

/usr/old/cc −I/usr/old/include [*flags*] foo.c **−lm**

SEE ALSO

arcv(8), ar(1), cc(1), a.out(5), ar(5)

BUGS

NAME

pstat − print system facts

SYNOPSIS

/etc/pstat [**−aixptufT**] [suboptions] [file]

DESCRIPTION

Pstat interprets the contents of certain system tables. If *file* is given, the tables are sought there, otherwise in */dev/kmem.* The required namelist is taken from */vmunix.* 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)
P	resulted from demand-page-from-inode exec format (see *exec*(2))

DADDR Disk address in swap, measured in multiples of 512 bytes.

CADDR Head of a linked list of loaded processes using this text segment.

SIZE Size of text segment, measured in multiples of 512 bytes.

IPTR Core location of corresponding inode.

CNT Number of processes using this text segment.

CCNT Number of processes in core using this text segment.

−p Print process table for active processes with these headings:

LOC The core location of this table entry.

S Run state encoded thus:

0	no process
1	waiting for some event
3	runnable

	4	being created
	5	being terminated
	6	stopped under trace
F	Miscellaneous state variables, or-ed together (hexadecimal):	
	000001	loaded
	000002	the scheduler process
	000004	locked for swap out
	000008	swapped out
	000010	traced
	000020	used in tracing
	000040	locked in by *lock*(2).
	000080	in page-wait
	000100	prevented from swapping during *fork*(2)
	000200	gathering pages for raw i/o
	000400	exiting
	001000	process resulted from a *vfork*(2) which is not yet complete
	002000	another flag for *vfork*(2)
	004000	process has no virtual memory, as it is a parent in the context of *vfork*(2)
	008000	process is demand paging data pages from its text inode.
	010000	process has advised of anomalous behavior with *vadvise*(2).
	020000	process has advised of sequential behavior with *vadvise*(2).
	040000	process is in a sleep which will timeout.
	080000	a parent of this process has exited and this process is now considered detached.
	100000	process used some new signal primitives, i.e. *sigser*(3); more system calls will restart.
	200000	process is owed a profiling tick.

POIP number of pages currently being pushed out from this process.
PRI Scheduling priority, see *nice*(2).
SIGNAL Signals received (signals 1-32 coded in bits 0-31),
UID Real user ID.
SLP Amount of time process has been blocked.
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 page frame number of the first page of the 'u-area' of the process. If swapped out, the position in the swap area measured in multiples of 512 bytes.
RSS Resident set size — the number of physical page frames allocated to this process.
SRSS RSS at last swap (0 if never swapped).
SIZE Virtual size of process image (data+stack) in multiples of 512 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 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.
DISC Line discipline; blank is old tty OTTYDISC or "new tty" for NTTYDISC or "net" for NETLDISC (see *bk*(4)).

−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).

−s print information about swap space usage: the number of (1k byte) pages used and free is given as well as the number of used pages which belong to text images.

−T prints the number of used and free slots in the several system tables and is useful for checking to see how full system tables have become if the system is under heavy load. **−m** and **−g** flags print the multiplexor tables. These tables are rather difficult to explain. The potential explorer should examine the multiplexor code in the system.

FILES

/vmunix namelist
/dev/kmem default source of tables

SEE ALSO

ps(1), stat(2), filsys(5)
K. Thompson, *UNIX Implementation*

BUGS

It would be very useful if the system recorded "maximum occupancy" on the tables reported by −T; even more useful if these tables were dynamically allocated.

NAME

quot — summarize file system ownership

SYNOPSIS

/etc/quot [option] ... [filesystem]

DESCRIPTION

Quot prints the number of blocks in the named *filesystem* currently owned by each user. If no *filesystem* is named, a default name is assumed. The following options are available:

−n Cause the pipeline **ncheck filesystem | sort +0n | quot −n filesystem** to produce a list of all files and their owners.

−c Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.

−f Print count of number of files as well as space owned by each user.

FILES

Default file system varies with system.
/etc/passwd to get user names

SEE ALSO

ls(1), du(1)

BUGS

Holes in files are counted as if they actually occupied space.

NAME

rc – command script for auto-reboot and daemons

SYNOPSIS

/etc/rc

DESCRIPTION

Rc is the command script which controls the automatic reboot

When an automatic reboot is in progress, *rc* is invoked with the argument *autoboot* and runs a *fsck* with option **−p** to "preen" all the disks of minor inconsistencies resulting from the last system shutdown and to check for serious inconsistencies caused by hardware or software failure. If this auto-check and repair succeeds, then the second part of *rc* is run.

The second part of *rc*, which is run after a auto-reboot succeeds and also if *rc* is invoked when a single user shell terminates (see *init*(8)), starts all the daemons on the system, preserves editor files and clears the scratch directory **/tmp**.

SEE ALSO

init(8), reboot(8)

BUGS

NAME

reboot – UNIX bootstrapping procedures

SYNOPSIS

/etc/reboot [**-n**] [**-q**]

DESCRIPTION

UNIX is started by placing it in memory at location zero and transferring to zero. Since the system is not reenterable, it is necessary to read it in from disk or tape each time it is to be bootstrapped.

Rebooting a running system. When a UNIX is running and a reboot is desired, *shutdown*(8) is normally used. If there are no users then **/etc/reboot** can be used. Reboot causes the disks to be synced, and then a multi-user reboot (as described below) is initiated. This causes a system to be booted and an automatic disk check to be performed. If all this succeeds without incident, the system is then brought up for many users.

Options to reboot are:

-n option avoids the sync. It can be used if a disk or the processor is on fire. (It is no longer necessary to reboot after rebuilding the root file system.)

-q reboots quickly and ungracefully, without shutting down running processes first.

Power fail and crash recovery. Normally, the system will reboot itself at power-up or after crashes. Provided the auto-restart is enabled on the machine front panel, an automatic consistency check of the file systems will be performed then and unless this fails the system will resume multi-user operations.

Cold starts. These are processor type dependent. On an 11/780, there are two floppy files for each disk controller, both of which cause boots from unit 0 of the root file system of a controller located on mba0 or uba0. One gives a single user shell, while the other invokes the multi-user automatic reboot. Thus these files are HPS and HPM for the single and multi-user boot from MASSBUS RP06/RM03/RM05 disks, UPS and UPM for UNIBUS storage module controller and disks such as the EMULEX SC-21 and AMPEX 9300 pair, or HKS and HKM for RK07 disks.

Giving the command

    >>>BOOT HPM

Would boot the system from (e.g.) an RP06 and run the automatic consistency check as described in *fsck*(8). (Note that it may be necessary to type control-P to gain the attention of the LSI-11 before getting the >>> prompt.) The command

    >>>BOOT ANY

invokes a version of the boot program in a way which allows you to specify any system as the system to be booted. It reads from the console a device specification (see below) followed immediately by a pathname.

On an 11/750, the reset button will boot from the device selected by the front panel boot device switch. In systems with RK07's, position B normally selects the RK07 for boot. This will boot multi-user. To boot from RK07 with boot flags you may specify

    >>>B/*n* DMA0

where, giving a *n* of 1 causes the boot program to ask for the name of the system to be bootstrapped, giving a *n* of 2 causes the boot program to come up single user, and a *n* of 3 causes both of these actions to occur.

The 11/750 boot procedure uses the boot roms to load block 0 off of the specified device. The /usr/mdec directory contains a number of bootstrap programs for the various disks which should be placed in a new pack via

cp /usr/mdec/*xx*boot /dev/*xx*?a

whenever a new bootable pack is to be created.

On both processors, the *boot* program finds the corresponding file on the given device, loads that file into memory location zero, and starts the program at the entry address specified in the program header (after clearing off the high bit of the specified entry address.) Normal line editing characters can be used in specifying the pathname.

If you have an rp06, rm05 or rm03 disk and wish to boot off of a file system which starts at cylinder 0 of unit 0, you can type "hp(0,0)vmunix" to the boot prompt; "up(0,0)vmunix" would specify a UNIBUS ampex 9300 drive, "rk(0,0)vmunix" would specify a RK-07 disk drive.

A device specification has the following form:

device(unit, minor)

where *device* is the type of the device to be searched, *unit* is 8* the mba or uba number plus the unit number of the device, and *minor* is the minor device index. The following list of supported devices may vary from installation to installation:

hp	RP06, RM03, RM05, RP07 or RM80 on MASSBUS
up	storage module drive on UNIBUS
ht	TE16,TU45,TU77 on MASSBUS
hk	RK07 on UNIBUS
tm	TM11 emulation tape drives on UNIBUS
ts	TS11 on UNIBUS

For tapes, the minor device number gives a file offset.

In an emergency, the bootstrap methods described in the paper "Setting up the Fourth Berkeley Software Tape" can be used to boot from a distribution tape.

FILES

/vmunix	system code
/boot	system bootstrap

SEE ALSO

crash(8), fsck(8), init(8), rc(8), shutdown(8), halt(8)

NAME

renice — alter priority of running process by changing nice

SYNOPSIS

/etc/renice pid [priority]

DESCRIPTION

Renice can be used by the super-user to alter the priority of a running process. By default, the nice of the process is made "19" which means that it will run only when nothing else in the system wants to. This can be used to nail long running processes which are interfering with interactive work.

Renice can be given a second argument to choose a nice other than the default. Negative nices can be used to make things go very fast.

FILES

/vmunix
/dev/kmem

SEE ALSO

nice(1)

BUGS

If you make the nice very negative, then the process cannot be interrupted. To regain control you must put the nice back (e.g. to 0.)

NAME

restor — incremental file system restore

SYNOPSIS

/etc/restor key [argument ...]

DESCRIPTION

Restor is used to read magtapes dumped with the *dump* command. The *key* specifies what is to be done. *Key* is one of the characters **rRxt** optionally combined with **f**.

f Use the first *argument* as the name of the tape instead of the default.

r or R The tape is read and loaded into the file system specified in *argument*. This should not be done lightly (see below). If the key is **R** *restor* asks which tape of a multi volume set to start on. This allows restor to be interrupted and then restarted (an *icheck* −s must be done before restart).

x Each file on the tape named by an *argument* is extracted. The file extracted is placed in a file with a numeric name supplied by *restor* (actually the inode number). In order to keep the amount of tape read to a minimum, the following procedure is recommended:

Mount volume 1 of the set of dump tapes.

Type the *restor* command.

Restor will announce whether or not it found the files, give the number it will name the file, and rewind the tape.

It then asks you to 'mount the desired tape volume'. Type the number of the volume you choose. On a 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 a later tape, thus the reverse order) and doesn't read through the tape if no files are. If you are working with a single volume dump or the number of files being restored is large, respond to the query with '1' and *restor* will read the tapes in sequential order.

If you have a hierarchy to restore you can use *dumpdir*(8) to produce the list of names and a shell script to move the resulting files to their homes.

t Print the date the tape was written and the date the filesystem was dumped from.

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

```
/etc/mkfs /dev/rrp0g 145673
restor r /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 unit varies with installation
rst*

SEE ALSO

dump(8), mkfs(8), dumpdir(8)

DIAGNOSTICS

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

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

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *restor* doesn't use it.

NAME

sa, accton − system accounting

SYNOPSIS

/etc/sa [−**abcdDfijkKlnrstuv**] [file]

/etc/accton [file]

DESCRIPTION

With an argument naming an existing *file, accton* causes system accounting information for every process executed to be placed at the end of the file. If no argument is given, accounting is turned off.

Sa reports on, cleans up, and generally maintains accounting files.

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

If a file name is given as the last argument, that file will be treated as the accounting file; */usr/adm/acct* is the default.

Output fields are labelled: "cpu" for the sum of user+system time (in minutes), "re" for real time (also in minutes), "k" for cpu-time averaged core usage (in 1k units), "avio" for average number of i/o operations per execution. With options fields labelled "tio" for total i/o operations, "k*sec" for cpu storage integral (kilo-core seconds), "u" and "s" for user and system cpu time alone (both in minutes) will sometimes appear.

There are near a googol of options:

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

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

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

d	Sort by average number of disk i/o operations.

D	Print and sort by total number of disk i/o operations.

f	Force no interactive threshold compression with −v flag.

i	Don't read in summary file.

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

k	Sort by cpu-time average memory usage.

K	Print and sort by cpu-storage integral.

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

m	Print number of processes and number of CPU minutes for each user.

n	Sort by number of calls.

r	Reverse order of sort.

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

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

u	Superseding all other flags, print for each command in the accounting file the user ID and command name.

v Followed by a number *n*, types the name of each command used *n* times or fewer. Await a reply from the terminal; if it begins with 'y', add the command to the category '**junk**.' This is used to strip out garbage.

FILES

/usr/adm/acct	raw accounting
/usr/adm/savacct	summary
/usr/adm/usracct	per-user summary

SEE ALSO

ac(8), acct(2)

BUGS

The number of options to this program is absurd.

NAME

savecore − save a core dump of the operating system

SYNOPSIS

savecore *dirname*

DESCRIPTION

Savecore is meant to be called at the end of the /etc/rc file. Its function is to save the core dump of the system (assuming one was made) and to write a reboot message in the shutdown log.

Savecore checks the core dump to be certain it corresponds with the current running unix. If it does it saves the core image in the file *dirname/*vmcore.n and it's brother, the namelist, *dirname/*vmunix.n The trailing ".n" in the pathnames is replaced by a number which grows every time *savecore* is run in that directory.

Before savecore writes out a core image, it reads a number from the file *dirname/*minfree. If there are fewer free blocks on the filesystem which contains *dirname* than the number obtained from the minfree file, the core dump is not done. If the minfree file does not exist, savecore always writes out the core file (assuming that a core dump was taken).

Savecore also writes a reboot message in the shut down log. If the system crashed as a result of a panic, *savecore* records the panic string in the shut down log too.

FILES

/usr/adm/shutdownlog Shut down log
/vmunix Current UNIX

BUGS

NAME

shutdown − close down the system at a given time

SYNOPSIS

/etc/shutdown [**−k**] [**−r**] [**−h**] time [warning-message ...]

DESCRIPTION

Shutdown provides an automated shutdown procedure which a super-user can use to notify users nicely when the system is shutting down, saving them from system administrators, hackers, and gurus, who would otherwise not bother with niceties.

Time is the time at which *shutdown* will bring the system down and may take two formats: +number and hour:min. The first form brings the system down in *number* minutes and the second brings the system down at the time of day indicated (as a 24−hour clock).

At intervals which get closer together as apocalypse approaches, warning messages are displayed at the terminals of all users on the system. Five minutes before shutdown, or immediately if shutdown is in less than 5 minutes, logins are disabled by creating /etc/nologin and writing a message there. If this file exists when a user logs in, *login*(1) prints its contents and exits. The file is removed just before *shutdown* exits.

At shutdown time a message is written in the file /usr/adm/shutdownlog, containing the time of shutdown, who ran shutdown and the reason. Then a terminate signal is sent at *init* to bring the system down to single-user state. Alternatively, if **-r**, **-h**, or **-k** was used, then *shutdown* will exec *reboot*(8), *halt*(8), or avoid shutting the system down (respectively). (If it isn't obvious, **−k** is to make people *think* the system is going down!)

The time of the shutdown and the warning message are placed in /etc/nologin and should be used to inform the users about when the system will be back up and why it is going down (or anything else).

FILES

/etc/nologin tells login not to let anyone log in
/usr/adm/shutdownlog log file for succesful shutdowns.

SEE ALSO

login(1), reboot(8)

BUGS

Only allows you to kill the system between now and 23:59 if you use the absolute time for shutdown.

Times to shutdown are not nice and round, i.e. "shutdown in 18 seconds".

NAME

sticky — executable files with persistent text

DESCRIPTION

While the 'sticky bit', mode 01000 (see *chmod*(2)), is set on a sharable executable file, the text of that file will not be removed from the system swap area. Thus the file does not have to be fetched from the file system upon each execution. As long as a copy remains in the swap area, the original text cannot be overwritten in the file system, nor can the file be deleted. (Directory entries can be removed so long as one link remains.)

Sharable files are made by the **−n** and **−z** options of *ld*(1).

To replace a sticky file that has been used do: (1) Clear the sticky bit with *chmod*(1). (2) Execute the old program to flush the swapped copy. This can be done safely even if others are using it. (3) Overwrite the sticky file. If the file is being executed by any process, writing will be prevented; it suffices to simply remove the file and then rewrite it, being careful to reset the owner and mode with *chmod* and *chown*(2). (4) Set the sticky bit again.

Only the super-user can set the sticky bit.

BUGS

Are self-evident.

Is largely unnecessary on the VAX; matters only for large programs that will page heavily to start, since text pages are normally cached incore as long as possible after all instances of a text image exit.

NAME

swapon − specify additional device for paging and swapping

SYNOPSIS

/etc/swapon −a
/etc/swapon name ...

DESCRIPTION

Swapon is used to specify additional devices on which paging and swapping are to take place. The system begins by swapping and paging on only a single device so that only one disk is required at bootstrap time. Calls to *swapon* normally occur in the system multi-user initialization file */etc/rc* making all swap devices available, so that the paging and swapping activity is interleaved across several devices.

Normally, the **−a** argument is given, causing all devices marked as ''sw'' swap devices in **/etc/fstab** to be made available.

The second form gives individual block devices as given in the system swap configuration table. The call makes only this space available to the system for swap allocation.

SEE ALSO

swapon(2), init(8)

FILES

/dev/[ru][pk]?b normal paging devices

BUGS

There is no way to stop paging and swapping on a device. It is therefore not possible to make use of devices which may be dismounted during system operation.

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 in a way not suitably done by *reboot*(8) or *halt*(8).

See *sync*(2) for details on the system primitive.

SEE ALSO

sync(2), halt(8), reboot(8), update(8)

NAME

update — periodically update the super block

SYNOPSIS

/etc/update

DESCRIPTION

Update is a program that executes the *sync*(2) primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

SEE ALSO

sync(2), sync(1), init(8)

BUGS

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync*(1) temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.

NAME

vipw – edit the password file with vi

SYNOPSIS

vipw

DESCRIPTION

Vipw edits the password file while setting the appropriate locks, and does any necessary processing after the password file is unlocked. If the password file is already being edited, then you will be told to try again later

SEE ALSO

chfn(1), chsh(1), passwd(1), passwd(5), adduser(8)

FILES

/etc/vipw.lock

BUGS

Vipw does not remove the vipw.lock file; this is not a bug, but people tend to think it is.

No one deals with left-over /etc/ptmp (the real lock) files after a system crash.

NAME

vpac — print raster printer/ploter accounting information

SYNOPSIS

/etc/vpac [**−W**] [**−s**] [**−r**] [**−t**] [name ...]

DESCRIPTION

Vpac reads the raster printer/plotter accounting files, accumulating the number of pages (for narrow fan-fold devices) or feet (for wide, roll paper devices) of paper consumed by each user, and printing out how much each user consumed in pages or feet and dollars (billed at 2 cents / page or 8 cents / foot). If any *names* are specified, then statistics are only printed for those users; usually, statistics are printed for every user who has used any paper.

The **−W** flag causes accounting to be done for a wide roll paper device. The default is to do accounting for a narrow, fan-fold device. The **−t** flag causes the output to be sorted by feet of paper; usually the output is sorted alphabetically by name. The **−r** flag reverses the sorting order. The **−s** flag causes the accounting information to be summarized on the summary accounting file; this summarization is necessary since on a busy system, the accounting file can grow by several lines per day.

FILES

/usr/adm/v?acct	raw accounting files
/usr/adm/v?_sum	summary accounting files

BUGS

The relationship between the computed price and reality is as yet unknown.