PLEXUS

Sys3 UNIX Programmer's Manual -- Vol 1B

98-05046.7 April 6, 1984

PLEXUS

Sys3 UNIX vol 1B

Programmer's Manual

Sys3 UNIX Programmer's Manual -- vol 1B

98-05046.7 April 6, 1984

PLEXUS COMPUTERS INC

2230 Martin Ave

Santa Clara, CA 95050

408/988-1755

The information contained herein is subject to change without notice. Therefore, Plexus Computers Inc. assumes no responsibility for the accuracy of the information presented in this document beyond its current release date.

Printed in the United States of America

REVISION RECORD

Plexus Sys3 UNIX Programmer's Manual -- vol 1B

REVISION LEVEL	DATE	DESCRIPTION
98-05046.1	December 23, 1982	First edition
98-05046.2	January 15, 1983	Editorial changes
98-05046.3	May 6, 1983	Removed unsupported graphics utilities; other small editorial changes
98-05046.4	June 20, 1983	Changes for Sys3 Rev. 3.0 (MC68000 version)
98-05046.5	August 19, 1983	New pages for the Plexus Network Operating System (NOS); other small additions and corrections
98-05046.6	November 18, 1983	Several new pages; other small additions and corrections
98-05046.7	April 6, 1984	First typeset version; additions and corrections

ACKNOWLEDGEMENTS

The form and much of the content of this manual come from the *UNIX Programmer's Manual Release 3.0* (Volume 1), edited by T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli.

PLEXUS INTRODUCTION

This release of the *Plexus Sys3 UNIX Programmer's Manual* is designed for use with Plexus Sys3. This manual includes a number of commands that are not part of stock SYSTEM III, plus enhancements to SYSTEM III commands. The majority of these are in Section 1 ("Commands and Application Programs"). Therefore, Volume 1 was separated into two different physical volumes. Section 1 is now in physical Volume 1A, and Sections 2 through 8 are in Volume 1B.

Some SYSTEM III commands are designed for use with UNIX systems on specific hardware such as the PDP-11; these commands are inappropriate for use on Plexus systems, and are thus not supported by Plexus. No source was provided for other SYSTEM III commands. The following table lists all the SYSTEM III commands that are not supported by Plexus, along with codes indicating why Plexus does not support them. The codes have the following meanings:

NA - Applicable to other hardware.
NI - Not implemented.
NS - No source available.

Command	Function	Code
as.pdp	assembler for PDP-11	NA
as.vax	assembler for VAX-11/780	NA
chess	the game of chess	NS
dj	DJ-11 asynchronous multiplexor	NA
dmc	communications link with built-in DDCMP protocol	NA
dn	DN-11 ACU interface	NA
dpr	off-line print	NA
dqs	DQS-11 interface for two-point BSC	NA
du	DU-11 synchronous line interface	NA
dz	DZ-11, DZ-11/KMC-11, DH-11 asynchronous multiplexors	NA
etp	Equipment Test Package	NA
fget	retrieve files from the HONEYWELL 6000	NA
fget.demon	file retrieval daemons	NA
fptrap	floating point interpreter	NA
fscv	convert files between PDP-11 and VAX-11/780 systems	NA
fsend	send files to the HONEYWELL 6000	NA
gcat	send phototypesetter output to the HONEYWELL 6000	NA
gcosmail	send mail to HIS user	NA
gdev	graphical device routines and filters	NI
ged	graphical editor	NI
gps	format of graphical files	NI
graphics	access graphical and numerical commands	NI
gutil	graphical utilities	NI
hasp	RJE (Remote Job Entry) to IBM	NA
hp	RP04/RP05/RP06 moving-head disk	NA
hs	RH11/RJS03-RJS04 fixed-head disk file	NA
ht	TU16 magnetic tape interface	NA
kas	assembler for the KMC11 microprocessor	NA
kl	KL-11 or DL-11 asynchronous interface	NA

kmc	KMC11 microprocessor	NA
kun	un-assembler for the KMC11/DMC11 microprocessor	NA
maze	generate a maze	NS
pcl	parallel communications link interface	NA
reversi	a game of dramatic reversals	NS
rf	RF11/RS11 fixed-head disk file	NA
rk	RK-11/RK03 or RK05 disk	NA
rl	RL-11/RL01 disk	NA
rp	RP-11/RP03 moving-head disk	NA
sdb	symbolic debugger	NA
sky	obtain ephemerides	NS
st	synchronous terminal interface	NA
stat	statistical network for graphical commands	NI
tm	TM11/TU10 magnetic tape interface	NA
toc	graphical table of contents routines	NI
vaxops	VAX-11/780 console operations	NA
vlx	VAX-11/780 LSI console floppy interface	NA

See the Introductions to each section for information on new commands.

BELL INTRODUCTION

(This Introduction was written by Bell Laboratories for the *UNIX User's Manual Release 1.0.*)

This manual describes the features of UNIX. It provides neither a general overview of UNIX (for that, see "The UNIX Time-Sharing System," *BSTJ*, Vol. 57, No. 6, Part 2, pp. 1905-29, by D. M. Ritchie and K. Thompson), nor details of the implementation of the system (see "UNIX Implementation," *BSTJ*, same issue, pp. 1931-46).

Not all commands, features, and facilities described in this manual are available in every UNIX system; for example, *yacc*(1) is usually not available in a UNIX system running on a PDP-11/23. When in doubt, consult your system's administrator.

This manual is divided into eight sections, some containing inter-filed sub-classes:

1. Commands and Application Programs:
   - 1. General-Purpose Commands.
   - 1C. Communications Commands.
   - 1G. Graphics Commands.
   - 1M. System Maintenance Commands.
2. System Calls.
3. Subroutines:
   - 3C. C and Assembler Library Routines.
   - 3M. Mathematical Library Routines.
   - 3S. Standard I/O Library Routines.
   - 3X. Miscellaneous Routines.
4. Special Files.
5. File Formats.
6. Games.
7. Miscellaneous Facilities.
8. System Maintenance Procedures.

Section 1 (*Commands and Application Programs*) describes programs intended to be invoked directly by the user or by command language procedures, as opposed to subroutines, which are intended to be called by the user's programs. Commands generally reside in the directory **/bin** (for **bin**ary programs). Some programs also reside in **/usr/bin**, to save space in **/bin**. These directories are searched automatically by the command interpreter called the *shell*. Sub-class 1C contains communication programs such as *cu*, *dpr*, etc. These entries may differ from system to system. Sub-class 1M contains system maintenance programs such as *fsck*, *mkfs*, etc., which generally reside in the directory **/etc**; these commands are not intended for use by the ordinary user due to their privileged nature. Some UNIX systems have a directory called **/usr/lbin**, containing local commands.

Section 2 (*System Calls*) describes the entries into the UNIX supervisor, including the C language interface.

Section 3 (*Subroutines*) describes the available subroutines. Their binary versions reside in various system libraries in the directories **/lib** and **/usr/lib**. See *intro*(3) for descriptions of these libraries and the files in which they are stored.

Section 4 (*Special Files*) discusses the characteristics of each system file that actually refers to an input/output device. The names in this section generally refer to the Digital Equipment Corporation's device names for the hardware, rather than to the names of the special files themselves.

Section 5 (*File Formats*) documents the structure of particular kinds of files; for example, the format of the output of the link editor is given in *a.out*(5). Excluded are files used by only one command (for example, the assembler's intermediate files). In general, the C language **struct** declarations corresponding to these formats can be found in the directories **/usr/include** and **/usr/include/sys**.

Section 6 (*Games*) describes the games and educational programs that, as a rule, reside in the directory **/usr/games**.

Section 7 (*Miscellaneous Facilities*) contains a variety of things. Included are descriptions of character sets, macro packages, etc.

Section 8 (*System Maintenance Procedures*) discusses crash recovery and boot procedures, etc. Information in this section is not of great interest to most users.

Each section consists of a number of independent entries of a page or so each. The name of the entry appears in the upper corners of its pages. Entries within each section are alphabetized, with the exception of the introductory entry that begins each section. The page numbers of each entry start at 1. Some entries may describe several routines, commands, etc. In such cases, the entry appears only once, alphabetized under its "major" name.

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

The **NAME** part gives the name(s) of the entry and briefly states its purpose.

The **SYNOPSIS** part summarizes the use of the program being described. A few conventions are used, particularly in Section 1 (*Commands*):

Boldface strings are literals and are to be typed just as they appear.

Italic strings usually represent substitutable argument prototypes and program names found elsewhere in the manual (they are underlined in the typed version of the entries).

Square brackets **[]** around an argument prototype indicate that the argument is optional. When an argument prototype is given as "name" or "file", it always refers to a *file* name.

Ellipses **...** are used to show that the previous argument prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus —, plus +, or equal sign = is often taken to be some sort of flag argument, even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with —, +, or =.

The **DESCRIPTION** part discusses the subject at hand.

The **EXAMPLE(S)** part gives example(s) of usage, where appropriate.

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

The **SEE ALSO** part gives pointers to related information.

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

The **WARNINGS** part points out potential pitfalls.

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

A table of contents and a permuted index derived from that table precede Section 1. On each *index* line, the title of the entry to which that line refers is followed by the appropriate section number in parentheses. This is important because there is considerable duplication of names among the sections, arising principally from commands that exist only to exercise a particular system call. On most systems, all entries are available on-line via the *man*(1) command, q.v.

HOW TO GET STARTED

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

Logging in. You must dial up UNIX from an appropriate terminal. UNIX supports full-duplex ASCII terminals. You must also have a valid user name, which may be obtained (together with the telephone number(s) of your UNIX system) from the administrator of your system. Common terminal speeds are 10, 15, 30, and 120 characters per second (110, 150, 300, and 1,200 baud); occasionally, speeds of 240, 480, and 960 characters per second (2,400, 4,800, and 9,600 baud) are also available. On some UNIX systems, there are separate telephone numbers for each available terminal speed, while on other systems several speeds may be served by a single telephone number. In the latter case, there is one "preferred" speed; if you dial in from a terminal set to a different speed, you will be greeted by a string of meaningless characters (the **login:** message at the wrong speed). Keep hitting the "break" or "attention" key until the **login:** message appears. Hard-wired terminals usually are set to the correct speed.

Most terminals have a speed switch that should be set to the appropriate speed and a half-/full-duplex switch that should be set to full-duplex. When a connection (at the speed of the terminal) has been established, the system types **login:** and you then type your user name followed by the "return" key. If you have a password (and you should!), the system asks for it, but does not print ("echo") it on the terminal. After you have logged in, the "return", "new-line", and "line-feed" keys will give exactly the same result.

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

For more information, consult *login*(1) and *getty*(8), which discuss the login sequence in more detail, and *stty*(1), which tells you how to describe the characteristics of your terminal to the system (*profile*(5) explains how to accomplish this last task automatically every time you log in).

Logging out. There are two ways to log out:

1. You can simply hang up the phone.
2. You can log out by typing an end-of-file indication (ASCII **EOT** character, usually typed as "control-d") to the shell. The shell will terminate and the **login:** message will appear again.

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

UNIX terminal input/output is full-duplex. It has full read-ahead, which means that you can type at any time, even while a program is typing at you. Of course, if you type during output, the output will have interspersed in it the input characters. However, whatever you type will be saved and interpreted in the correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away *all* the saved characters.

On an input line from a terminal, the character @ "kills" all the characters typed before it. The character # erases the last character typed. Successive uses of # will erase characters back to, but not beyond, the beginning of the line; @ and # can be typed as themselves by preceding them with \ (thus, to erase a \, you need two #s). These default erase and kill characters can be changed; see *stty*(1).

The ASCII **DC3** (control-s) character can be used to temporarily stop output. It is useful with CRT terminals to prevent output from disappearing before it can be read. Output is resumed when a **DC1** (control-q) or a second **DC3** (or any other character, for that matter) is typed. The **DC1** and **DC3** characters are not passed to any other program when used in this manner.

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

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

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

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

How to run a program. When you have successfully logged into UNIX, a program called the shell is listening to your terminal. The shell reads the lines you type, splits them into a command name and its arguments, and executes the command. A command is simply an executable program. Normally, the shell looks first in your current directory (see *The current directory* below) for a program with the given name, and if none is there, then in system directories. There is nothing special about system-provided commands except that they are kept in directories where the shell can find them. You can also keep commands in your own directories and arrange for the shell to find them there.

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

When a program terminates, the shell will ordinarily regain control and type a **$** at you to indicate that it is ready for another command. The shell has many other capabilities, which are described in detail in *sh*(1).

The current directory. UNIX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he or she also created a directory for you (ordinarily with the same name as your user name, and known as your *login* or *home* directory). When you log in, that directory becomes your *current* or *working* directory, and any file name you type is by default assumed to be in that directory. Because you are the owner of this directory, you have full permissions to read, write, alter, or destroy its contents. Permissions to have your will with other directories and files will have been granted or denied to you by their respective owners, or by the system administrator. To change the current directory 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 /, which is the name of the *root* directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory (followed by a /), until finally the file name is reached (e.g., **/usr/ae/filex** refers to file **filex** in directory **ae**, while **ae** is itself a subdirectory of **usr**; **usr** springs directly from the root directory). See *intro*(2) for a formal definition of *path name*.

If your current directory contains subdirectories, the path names of files therein begin with the name of the corresponding subdirectory (*without* a prefixed /). Without important exception, a path name may be used anywhere a file name is required.

Important commands that modify the contents of files are *cp*(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). Use *mkdir*(1) for making directories and *rmdir*(1) for destroying them.

For a fuller discussion of the file system, see the references cited at the beginning of the *INTRODUCTION* above. 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 *ed*(1). The four principal languages available under UNIX are C (see *cc*(1)), Fortran (see *f77*(1)), bs (a compiler/interpreter in the spirit of Basic, see *bs*(1)), and assembly language (see *as*(1)). After the program text has been entered with the editor and written into a file (whose name has the appropriate suffix), you can give the name of that file to the appropriate language processor as an argument. Normally, the output of the language processor will be left in a file in the current directory named **a.out** (if that output is precious, use *mv*(1) to give it a less vulnerable name). If the program is written in assembly language, you will probably need to load with it library subroutines (see *ld*(1)). Fortran and C call the loader automatically; programs written in *bs*(1) are interpreted and, therefore, do not need to be loaded.

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

If any execution (run-time) errors occur, you will need *adb*(1) to examine the remains of your program.

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 *ed*(1). The commands most often used to write text on a terminal are *cat*(1), *pr*(1), and *nroff*(1). The *cat*(1) command simply dumps ASCII text on the terminal, with no processing at all. The *pr*(1) command paginates the text, supplies headings, and has a facility for multi-column output. *Nroff*(1) is an elaborate text formatting program, and requires careful forethought in entering both the text and the formatting commands into the input file; it produces output on a typewriter-like terminal. *Troff*(1) is very similar to *nroff*(1), but produces its output on a phototypesetter (it was used to typeset this manual). There are several "macro" packages (especially the so-called *mm* package) that significantly ease the effort required to use *nroff*(1) and *troff*(1); Section 7 entries for these packages indicate where you can find their detailed descriptions.

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 or she next logs in. The corresponding entries in this manual also suggest how to respond to these two commands if you are their target.

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

CONTENTS

1. Commands and Application Programs

2. System Calls

3. Subroutines

4. Special Files

5. File Formats

6. Games

7. Miscellaneous Facilities

8. System Maintenance Procedures

PERMUTED INDEX

NAME

intro - introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always -1; the individual descriptions 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 been indicated.

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in **<error.h>**.

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
: No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

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. 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 on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
: An argument list longer than 5,120 bytes is presented to a member of the *exec* family.

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 (respectively write) request is made to a file which is open only for writing (respectively reading).

10 ECHILD No child processes
: A *wait*, was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
: A *fork*, failed because the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough space
During an *exec*, *brk*, or *sbrk*, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*.

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 use an argument of a system call.

15 ENOTBLK Block device required
A non-block 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 (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled.

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 prefix or as an argument to *chdir*(2).

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*, or *kill*; reading or writing a file for which *lseek* has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual. This error occurs if an *open* of a serial port, e.g., **/dev/console** or **/dev/ttyx**, would exceed the maximum allowable (usually 16 or 32).

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
No process may have more than 20 file descriptors open at a time.

25 ENOTTY Not a typewriter

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 file size (1,082,201,088 bytes) or ULIMIT; see

ulimit(2).

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.

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 the maximum number of links (1000) 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 not representable within machine precision.

DEFINITIONS

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID

A new process is created by a currently active process; see *fork*(2). The parent process ID of a process is the process ID of its creator.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(2).

Tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(2) and *signal*(2).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec*(2).

Super-user

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Name.

Names consisting of up to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding 0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell. See *sh*(1).

Path Name and Path Prefix

A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

More precisely, a path name is a null-terminated character string constructed as follows:

```
<path-name>::=<file-name>|<path-prefix><file-name>|/
<path-prefix>::=<rtprefix>|/<rtprefix>
<rtprefix>::=<dirname>/|<rtprefix><dirname>/
```

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory.

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory.

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process's root directory need not be the root directory of the root file system.

File Access Permissions.

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The process's effective user ID is super-user.

The process's effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's group ID matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process's effective user ID does not match the user ID of the owner of the file, and the process's effective group ID does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

NOTES

Plexus adds the system calls *lockf* and *ugrow* and the header file *syscall*, which lists the numeric ids of system calls recognized by Plexus Sys3 UNIX. Plexus also adds *rmount* and *rumount*, for use with the Plexus Network Operating System (NOS).

SEE ALSO

intro(3).

NAME

access - determine accessibility of a file

SYNOPSIS

int access (path, amode)
char *path;
int amode;

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

Read, write, or execute (search) permission is requested for a null path name. [ENOENT]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), stat(2).

NAME

acct - enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(2). The effective user ID of the calling process must be super-user to use this call.

Path points to a path name naming the accounting file. The accounting file format is given in *acct*(5).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

Acct will fail if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

An attempt is being made to enable accounting when it is already enabled. [EBUSY]

A component of the path prefix is not a directory. [ENOTDIR]

One or more components of the accounting file's path name do not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The file named by *path* is not an ordinary file. [EACCES]

Mode permission is denied for the named accounting file. [EACCES]

The named file is a directory. [EISDIR]

The named file resides on a read-only file system. [EROFS]

Path points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

acct(1M), acct(5).

NAME

alarm - set a process's alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

Alarm instructs the calling process's alarm clock to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2).

Alarm requests are not stacked; successive calls reset the calling process's alarm clock.

If *sec* is 0, any previously made alarm request is canceled.

RETURN VALUE

Alarm returns the amount of time previously remaining in the calling process's alarm clock.

SEE ALSO

pause(2), signal(2).

NAME

brk, sbrk - change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases.

Brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

Brk and *sbrk* will fail without making any change in the allocated space if such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)). [ENOMEM]

RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2).

NAME

chdir - change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Path points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the path name. [EACCES]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chroot(2).

NAME

chown - change owner and group of a file

SYNOPSIS

int chown (path, owner, group)
char *path;
int owner, group;

DESCRIPTION

Path points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

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

Chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2).

NAME

chroot - change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. can not be used to access files outside the subtree rooted at the root directory.

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

Any component of the path name is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not super-user. [EPERM]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chdir(2).

NAME

close - close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*.

Close will fail if *fildes* is not a valid open file descriptor. [EBADF]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

creat - create a new file or rewrite an existing one

SYNOPSIS

int creat (path, mode)
char *path;
int mode;

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

Upon successful completion, a non-negative integer, namely the file descriptor, is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls. See *fcntl*(2). No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The path name is null. [ENOENT]

The file does not exist and the directory in which the file is to be created does not permit writing. [EACCES]

The named file resides or would reside on a read-only file system. [EROFS]

The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]

The file exists and write permission is denied. [EACCES]

The named file is an existing directory. [EISDIR]

Twenty (20) file descriptors are currently open. [EMFILE]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), dup(2), lseek(2), open(2), read(2), umask(2), write(2).

NAME

dup - duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer. (i.e., both file descriptors share one file pointer.)

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Twenty (20) file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

execl, execv, execle, execve, execlp, execvp - execute a file

SYNOPSIS

int execl (path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp);
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, 0)
char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)
char *file, *argv[ ];

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header (see *a.out*(5)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

Path points to a path name that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(7)). The environment is supplied by the shell (see *sh*(1)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(2).

If the set-user-ID mode bit of the new process file is set (see *chmod*(2)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

Profiling is disabled for the new process; see *profil*(2).

The new process also inherits the following attributes from the calling process:

nice value (see *nice*(2))
process ID
parent process ID
process group ID
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))

Exec will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's path name do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Search permission is denied for a directory listed in the new process file's path prefix. [EACCES]

The new process file is not an ordinary file. [EACCES]

The new process file mode denies execution permission. [EACCES]

The new process file has the appropriate access permission, but has an invalid magic number in its header. [ENOEXEC]

The new process file is a pure procedure (shared text) file that is currently open for writing by some process. [ETXTBSY]

The new process requires more memory than is allowed by the system-imposed maximum MAXMEM. [ENOMEM]

The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes. [E2BIG]

The new process file is not as long as indicated by the size values in its header. [EFAULT]

Path, *argv*, or *envp* point to an illegal address. [EFAULT]

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be -1 and *errno* will be set to indicate the error.

SEE ALSO

exit(2), fork(2).

NAME

exit - terminate process

SYNOPSIS

exit (status)
int status;

DESCRIPTION

Exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table, it has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times.*

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(2)) inherits each of these processes.

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(2).

If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each processes that has a process group ID equal to that of the calling process.

SEE ALSO

signal(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

fcntl - file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The *cmds* available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL Get *file* status flags.

F_SETFL Set *file* status flags to *arg*. Only certain flags can be set; see *fcntl*(7).

Fcntl will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Cmd is F_DUPFD and 20 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and *arg* is negative or greater than 20. [EINVAL]

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD A new file descriptor.
F_GETFD Value of flag (only the low-order bit is defined).
F_SETFD Value other than -1.
F_GETFL Value of file flags.
F_SETFL Value other than -1.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), exec(2), open(2), fcntl(7).

NAME

fork - create a new process

SYNOPSIS

int fork ()

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to **0**; see *times*(2).

Fork returns a value of **0** to the child process.

Fork returns the process ID of the child process to the parent process.

Fork will fail and no child process will be created if one or more of the following are true:

The system-imposed limit on the total number of processes under execution would be exceeded. [EAGAIN]

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. [EAGAIN]

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exec(2), wait(2).

NAME

getpid, getpgrp, getppid - get process, process group, and parent process IDs

SYNOPSIS

int getpid ()

int getpgrp ()

int getppid ()

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getuid, geteuid, getgid, getegid - get real user, effective user, real group, and effective group IDs

SYNOPSIS

int getuid ()

int geteuid ()

int getgid ()

int getegid ()

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl - control device

SYNOPSIS

#include <sys/ioctl.h>

ioctl(fildes, request, arg)

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.

Ioctl will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Fildes is not associated with a character special device. [ENOTTY]

Request or *arg* is not valid. See *tty*(4). [EINVAL]

RETURN VALUE

If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

tty(4).

NAME

kill - send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The effective user ID of the sending process must match the real user ID of the receiving process unless, the effective user ID of the sending process is super-user, or the process is sending to itself.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(2)) and will be referred to below as *proc0* and *proc1* respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not -1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be sent if one or more of the following are true:

> *Sig* is not a valid signal number. [EINVAL]
>
> No process can be found corresponding to that specified by *pid*. [ESRCH]
>
> The sending process is not sending to itself, its effective user ID is not super-user, and its effective user ID does not match the real user ID of the receiving process. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2).

NAME

link - link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link will fail and no link will be created if one or more of the following are true:

A component of either path prefix is not a directory. [ENOTDIR]

A component of either path prefix does not exist. [ENOENT]

A component of either path prefix denies search permission. [EACCES]

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not super-user. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

Path2 points to a null path name. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory on a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

link(1M), unlink(2).

NAME

lock - lock a process in memory

SYNOPSIS

lock(flag)

DESCRIPTION

If the *flag* argument is non-zero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is un*lock*ed. This call may be executed only by the super-user.

BUGS

*Lock*ed processes interfere with the compaction of primary memory and can cause deadlock. This system call is not considered a permanent part of the system.

NAME

lockf - provide exclusive file regions for reading or writing

SYNOPSIS

lockf (fildes, mode, size) long size;

DESCRIPTION

Lockf allows a specified number of bytes to be accessed only by the *lockf* process. Other processes that attempt to lock, read, or write the locked area must sleep until the area becomes unlocked.

Fildes is the word returned from a successful *open*, *creat*, *dup*, or *pipe* system call.

Mode is zero to unlock the area. Mode is 1 or 2 to lock the area. If the mode is 1, and the area has some other lock on it, then the process sleeps until the entire area is available. If the mode is 2, and the area is locked, an error is returned; otherwise the area is locked.

Size is the number of contiguous bytes to be locked or unlocked. The area to be locked starts at the current offset in the file. If *size* is 0, the area to end of file is locked.

Deadlock may occur when a process controlling a locked area is put to sleep at the same time it is accessing another process's locked area. Thus calls to *lockf*, *read*, or *write* scan for a deadlock prior to sleeping on a locked area. An error return is made if sleeping on the locked error would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked area for the same process. When this or adjacent areas occur, the areas are combined into a single area. Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process.

While locks may be applied to special files or pipes, read/ write operations will not be blocked. Closing *fildes* automatically releases any locks the process has on a file.

NOTES

This is a Plexus command. It is not part of standard SYSTEM III.

SEE ALSO

open(2), creat(2), read(2), write(2), dup(2), close(2).

DIAGNOSTICS

The value -1 is returned if the file does not exist, or if a deadlock using file locks would occur. EACCES is returned for lock requests in which the area is already locked by another process. EDEADLOCK is returned by *locking*, *read*, or *write* if a deadlock would occur. EDEADLOCK will also be returned when the locktable overflows.

NAME

locking - provide exclusive file regions for reading or writing

SYNOPSIS

locking (fildes, mode, size) long size;

DESCRIPTION

Locking allows a specified number of bytes to be accessed only by the *locking* process. Other processes that attempt to lock, read, or write the locked area must sleep until the area becomes unlocked.

Fildes is the word returned from a successful *open*, *creat*, *dup*, or *pipe* system call.

Mode is zero to unlock the area. Mode is 1 or 2 to lock the area. If the mode is 1, and the area has some other lock on it, then the process sleeps until the entire area is available. If the mode is 2, and the area is locked, an error is returned; otherwise the area is locked.

Size is the number of contiguous bytes to be locked or unlocked. The area to be locked starts at the current offset in the file. If *size* is 0, the area to end of file is locked.

Deadlock may occur when a process controlling a locked area is put to sleep at the same time it is accessing another process's locked area. Thus calls to *locking*, *read*, or *write* scan for a deadlock prior to sleeping on a locked area. An error return is made if sleeping on the locked error would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked area for the same process. When this or adjacent areas occur, the areas are combined into a single area. Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process.

While locks may be applied to special files or pipes, read/ write operations will not be blocked. Closing *fildes* automatically releases any locks the process has on a file.

NOTES

This is a Plexus command. It is not part of standard SYSTEM III.

SEE ALSO

open(2), creat(2), read(2), write(2), dup(2), close(2).

DIAGNOSTICS

The value -1 is returned if the file does not exist, or if a deadlock using file locks would occur. EACCES is returned for lock requests in which the area is already locked by another process. EDEADLOCK is returned by *locking*, *read*, or *write* if a deadlock would occur. EDEADLOCK will also be returned when the locktable overflows.

NAME

lseek - move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* 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*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

Whence is not 0, 1 or 2. [EINVAL and SIGSYS signal]

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

mknod - make a directory, or a special or ordinary file

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*, where the value of *mode* is interpreted as follows:

- 0170000 file type (S_IFMT); one of the following:
  - 0010000 fifo special (S_IFIFO)
  - 0020000 character special (S_IFCHR)
  - 0040000 directory (S_IFDIR)
  - 0060000 block special (S_IFBLK)
  - 0100000 or 0000000 ordinary file (S_IFREG)
- 0004000 set user ID on execution (S_ISUID)
- 0002000 set group ID on execution (S_ISGID)
- 0001000 save text image after execution (S_ISVTX)
- 0000777 access permissions; constructed from the following
  - 0000400 read by owner (S_IREAD)
  - 0000200 write by owner (S_IWRITE)
  - 0000100 execute (search on directory) by owner (S_IEXEC)
  - 0000070 read, write, execute (search) by group
  - 0000007 read, write, execute (search) by others

Values of *mode* other than those above are undefined and should not be used.

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2). If *mode* indicates a block or character special file, *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Mknod may be invoked only by the super-user for file types other than FIFO special.

Mknod will fail and the new file will not be created if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(1), mknod(1M), chmod(2), exec(2), umask(2), fs(5).

NAME

mount - mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

Mount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

Dir is not a directory. [ENOTDIR]

Spec or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]

The device associated with *spec* is currently mounted. [EBUSY]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(1M), umount(2).

NAME
nice - change priority of a process

SYNOPSIS
int nice (incr)
int incr;

DESCRIPTION
Nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Nice will fail and not change the nice value if *incr* is negative and the effective user ID of the calling process is not super-user. [EPERM]

RETURN VALUE
Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO
nice(1), exec(2).

NAME

open - open for reading or writing

SYNOPSIS

#include <fcntl.h>
int open (path, oflag[, mode])
char *path;
int oflag, mode;

DESCRIPTION

Path points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

0 **O_RDONLY** Open for reading only.

1 **O_WRONLY** Open for writing only.

2 **O_RDWR** Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL The O_EXCL flag is undefined if O_CREAT is 0; it is defined only when O_CREAT is set. O_EXCL and O_CREAT both cause *open* to fail if the file exists. Use of both these flags allows a process to create a temporary file and know that it is the only cooperating process that has use of the file. O_EXCL does not grant exclusive use of an existing file. Also, another non-cooperating process can *open* the file without the O_EXCL bit.

Upon successful completion a non-negative integer, the file descriptor, is returned.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

No process may have more than 20 file descriptors open simultaneously.

The path must be non-null, i.e., 0 is illegal.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

Twenty (20) file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the process's allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

The maximum number of serial ports, e.g., **/dev/ttyx**, are currently open. [EINVAL]

RETURN VALUE

Upon successful completion, a non-negative integer, namely a file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NOTES

The EINVAL message is a Plexus addition.

SEE ALSO

close(2), creat(2), dup(2), fcntl(2), lseek(2), read(2), write(2).

NAME

pause - suspend process until signal

SYNOPSIS

pause ()

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal catching-function (see *signal*(2)), the calling process resumes execution from the point of suspension; with a return value of -1 from *pause* and *errno* set to EINTR.

RETURN VALUE

If no error, a value of 0 is returned.

SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

NAME

phys - allow a process to access physical memory

SYNOPSIS

long phys (virtualpage, pagecount, physaddr)
int virtualpage;
long pagecount;
long physpage;

DESCRIPTION

The argument *virtualpage* specifies a process (data-space) address range of *pagecount X 4K* bytes starting at virtual address *virtualpage X 4K* bytes. This address range is mapped into physical address *physpage X 4K* bytes. All three arguments, *virtualpage*, *pagecount*, and *physpage*, correspond to 4K (4096) byte pages, which is the logical and physical page size of the machine. If *pagecount* is zero, any previous mapping of *virtualpage* is nullified. If *pagecount* is -1, the previous logical to physical mapping for *virtualpage* is returned. (In the cases where *pagecount* is 0 or -1, *physaddr* is ignored.) For exmaple, the call

```
phys(0x10,2,0x100);
```

will map virtual addresses 0x10000-0x12000 to physical addresses 0x100000-0x102000.

This call may be executed only by the superuser.

RETURN VALUE

Upon successful completion, the previous page number associated with the logical page is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

syslock(2)

BUGS

If an error is encountered while changing the mapping, the mapping for the valid pages may be changed anyway.

This system call is obviously very machine-dependent and very dangerous. It was originally in VERSION 7 UNIX but was removed from SYSTEM III. It is not considered a permanent part of the system.

NAME

pipe - create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Writes up to 10240 bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

Pipe will fail if 19 or more file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

sh(1), read(2), write(2).

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 (50th second for the Z8000, 64th second for the MC68000); *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 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) 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 will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

NOTES

Plexus clock tick is each 50th second for the Z8000, each 64th second for the MC68000.

SEE ALSO

prof(1), monitor(3C).

NAME

ptrace - process trace

SYNOPSIS

int ptrace (request, pid, addr, data);
int request, pid, addr, data;

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *adb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated (as on the Z8000) request **1** returns a word from I space, and request **2** returns a word from D space. If I and D space are not separated (as on the 68000), either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses in this area range from 0 to 2048 on the Z8000, and 0 to 4096 on the 68000. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated (as on the Z8000), request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated (as on the 68000), either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are:

the general registers (registers 0-15)

the program counter and FCW

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request simulates a the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request **7**. The trace bit causes on interrupt upon completion of one machine instruction. This effectively allows single stepping of the child.
Note: the trace bit is turned off after an interrupt.

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**.

GENERAL ERRORS

Ptrace will in general fail if one or more of the following are true:

Request is an illegal number. [EIO]

Pid identifies a child that does not exist or has not executed a *ptrace* with request **0**. [ESRCH]

NOTES

Although functionally identical to the stock SYSTEM III system call, some architectural differences between Plexus and DEC hardware dictated a slightly modified implementation.

SEE ALSO

adb(1), exec(2), signal(2), wait(2).

NAME

read - read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *tty*(4)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data becomes available.

Read will fail if one or more of the following are true:

> *Fildes* is not a valid file descriptor open for reading. [EBADF]
>
> *Buf* points outside the allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned and *errno* is set to indicate the error.

NOTES

The Plexus ICP limits *nbyte* to 512 for TTY devices.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), open(2), pipe(2), tty(4).

NAME

rmount - mount a remote file system directory

SYNOPSIS

int rmount (rdir, node, dir, rwflag)
char *rdir, *node, *dir;
int rwflag;

DESCRIPTION

Rmount requests that a remote file system directory identified by *rdir*, at the remote system identified by *node*, be mounted on the directory identified by *dir*. *Rdir* and *dir* are pointers to path names. *Node* is a pointer to the remote system.

Upon successful completion, references to the file *dir* will refer to the specified directory *rdir* at the remote system node.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden; otherwise, writing is permitted according to individual file accessibility.

Rmount may be invoked only by the super-user.

Rmount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files or node names does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Dir or *rdir* is not a directory. [ENOTDIR]

Rdir, *node*, or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]

The remote directory *rdir* at the remote system *node* is currently rmounted. [EBUSY]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error. EIO is returned in *errno* if the *rmount* fails because of excessive timeouts.

NOTES

This command is available as part of the Plexus Network Operating System (NOS) only.

SEE ALSO

mount(1M), rmount(1M), umount(2), rumount(2).

NAME

rumount - unmount a remote file system directory

SYNOPSIS

int rumount (rdir, node)
char *rdir;
char *node;

DESCRIPTION

Rumount requests that a previously mounted remote file system directory *rdir* at the remote system *node* be unmounted. *Rdir* is a pointer to a path name. *Node* is a pointer to the remote system name. After unmounting the remote file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Rumount may be invoked only by the super-user.

Rumount will fail if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

Rdir or the remote system *node* does not exist. [ENXIO]

The remote file system directory *rdir* at the remote system *node* is not mounted. [EINVAL]

A file on *rdir* is busy locally. [EBUSY]

Rdir or *node* points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NOTES

This command is available as part of the Plexus Network Operating System (NOS) only.

SEE ALSO

mount(1M), rmount (1M), mount(2), rmount(2).

NAME

rmount - mount a remote file system directory

SYNOPSIS

int rmount (rdir, node, dir, rwflag)
char *rdir, *node, *dir;
int rwflag;

DESCRIPTION

Rmount requests that a remote file system directory identified by *rdir*, at the remote system identified by *node*, be mounted on the directory identified by *dir*. *Rdir* and *dir* are pointers to path names. *Node* is a pointer to the remote system.

Upon successful completion, references to the file *dir* will refer to the specified directory *rdir* at the remote system node.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if 1, writing is forbidden; otherwise, writing is permitted according to individual file accessibility.

Rmount may be invoked only by the super-user.

Rmount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files or node names does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Dir or *rdir* is not a directory. [ENOTDIR]

Rdir, *node*, or *dir* points outside the process's allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]

The remote directory *rdir* at the remote system *node* is currently rmounted. [EBUSY]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NOTES

This command is available as part of the Plexus Network Operating System (NOS) only.

SEE ALSO

mount(1M), rmount(1M), umount(2), rumount(2).

NAME

setpgrp - set process group ID

SYNOPSIS

int setpgrp ()

DESCRIPTION

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2).

NAME

setuid, setgid - set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

Setuid is used to set the real user ID and effective user ID of the calling process.

Setgid is used to set the real group ID and effective group ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

getuid(2), intro(2).

NAME

signal - specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	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
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	death of a child (see *WARNING* below)
SIGPWR	19	power fail (see *WARNING* below)

See below for the significance of the asterisk in the above list.

Func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL - terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with the following consequences:

All of the receiving process's open file descriptors will be closed.

If the parent process of the receiving process is executing a *wait*, it will be notified of the termination of the receiving process and the terminating signal's number will be made available to the parent process; see *wait*(2).

If the parent process of the receiving process is not executing a *wait*, the receiving process will be transformed into a zombie process (see *exit*(2) for definition of zombie process).

The parent process ID of each of the receiving process's existing child processes and zombie processes will be set to 1. This means the initialization process (see *intro*(2)) inherits each of these processes.

An accounting record will be written on the accounting file if the system's accounting routine is enabled; see *acct*(2).

If the receiving process's process ID, tty group ID, and process group ID are equal, the signal **SIGHUP** will be sent to all of the processes that have a process group ID equal to the process group ID of the receiving process.

A "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN - ignore signal
The signal *sig* is to be ignored.

Note: the signal **SIGKILL** cannot be ignored.

function address - catch signal
Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted and the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, **SIGCLD**, or **SIGPWR**.

When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call will return a -1 to the calling process with *errno* set to EINTR.

Note: the signal **SIGKILL** cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending **SIGKILL** signal.

Signal will fail if one or more of the following are true:

Sig is an illegal signal number, including **SIGKILL**. [EINVAL]

Func points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C).

WARNING

Two other signals that behave differently than the signals described above exist in this release of the system; they are:

SIGCLD	18	death of a child (not reset when caught)
SIGPWR	19	power fail (not reset when caught)

There is no guarantee that, in future releases of UNIX, these signals will continue to behave as described below; they are included only for compatibility with other versions of UNIX. Their use in new programs is strongly discouraged.

For these signals, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate; see *exit*(2).

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** except, that while the process is executing the signal-catching function any received **SIGCLD** signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

The **SIGCLD** affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NAME

stat, fstat - get file status

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;

DESCRIPTION

Path points to a path name naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort  st_mode;    /* File mode; see mknod(2) */
ino_t   st_ino;     /* Inode number */
dev_t   st_dev;     /* ID of device containing */
                    /* a directory entry for this file */
dev_t   st_rdev;    /* ID of device */
                    /* This entry is defined only for */
                    /* character special or block special files */
short   st_nlink;   /* Number of links */
ushort  st_uid;     /* User ID of the file's owner */
ushort  st_gid;     /* Group ID of the file's group */
off_t   st_size;    /* File size in bytes */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Time of last data modification */
time_t  st_ctime;   /* Time of last file status change */
                    /* Times measured in seconds since */
                    /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Stat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Buf or *path* points to an invalid address. [EFAULT]

Fstat will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Buf points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), time(2), unlink(2).

NAME

stime - set time

SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Stime will fail if the effective user ID of the calling process is not super-user. [EPERM]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(2).

NAME

sync - update super-block

SYNOPSIS

sync ( )

DESCRIPTION

Sync causes all information in 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 *fsck*, *df*, etc. It is mandatory before a boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

SEE ALSO

sync(1M).

NAME

syscall.h - numeric id of system call

SYNOPSIS

/usr/include/syscall.h

DESCRIPTION

The Plexus UNIX operating system recognizes the following system calls. They are used in the Z8000 sc xx assembly instruction.

```
#defineINDIR  0
#defineEXIT   1
#defineFORK   2
#defineREAD   3
#defineWRITE  4
#defineOPEN   5
#defineCLOSE  6
#defineWAIT   7
#defineCREAT  8
#defineLINK   9
#defineUNLINK        10
#defineEXEC   11
#defineCHDIR  12
#defineTIME   13
#defineMKNOD         14
#defineCHMOD         15
#defineCHOWN         16
#defineBREAK  17
#defineSTAT   18
#defineLSEEK  19
#defineGETPID        20
#defineMOUNT         21
#defineUMOUNT        22
#defineSETUID        23
#defineGETUID        24
#defineSTIME  25
#definePTRACE        26
#defineALARM  27
#defineFSTAT  28
#definePAUSE  29
#defineUTIME  30
#defineSTTY   31
#defineGTTY   32
#defineACCESS        33
#defineNICE   34
/*            35     Version 7: FTIME */
#defineSYNC   36
#defineKILL   37
#defineCSW    38     /* Not in Weco R-III */
#defineSETPGRP       39
#defineDUP    41
#definePIPE   42
#defineTIMES  43
#definePROFIL        44
#defineLOCKING       45       /* Not in Weco R-III */
#defineSETGID        46
```

```
#defineGETGID       47
#defineSIGNAL       48
#defineACCT  51
/*           52     Version 7: PHYS        */
/*           53     Version 7: LOCK        */
#defineIOCTL  54
#defineREBOOT       55      /* Not in Weco R-III */
#define FCNTL56     /* Version 7: MPX */
#define PWBSYS      57      /* UNAME and USTAT sys calls.  V7: undefined */
#defineEXECE 59
#defineUMASK60
#defineCHROOT       61
#define UGROW       62      /* R-III: FCNTL */
#define ULIMIT63    /* Version 7: undefined */
#define RMOUNT  64    /* NOS only */
#define RUMOUNT 65     /* NOS only */
```

NAME

time - get time

SYNOPSIS

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

Time will fail if *tloc* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

stime(2).

NAME

times - get process and child process times

SYNOPSIS

```
long times (buffer)
struct tbuffer *buffer;
struct tbuffer {
        long    utime;
        long    stime;
        long    cutime;
        long    cstime;
}
```

DESCRIPTION

Times fills the structure pointed to by *buffer* with time-accounting information. This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*.

All times are in 50ths of a second for the Z8000, 64ths of a second for the MC68000.

Utime is the CPU time used while executing instructions in the user space of the calling process.

Stime is the CPU time used by the system on behalf of the calling process.

Cutime is the sum of the *utimes* and *cutimes* of the child processes.

Cstime is the sum of the *stimes* and *cstimes* of the child processes.

Times will fail if *buffer* points to an illegal address. [EFAULT]

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time -- in 50ths of a second for the Z8000, 64ths of a second for the MC68000 -- since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a -1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

NAME

ugrow - change system stack limit

SYNOPSIS

ugrow(addr);
char *addr;
char *_endstk

DESCRIPTION

Ugrow sets the lower limit on the user's stack area. Pushing the stack to an address lower than this limit could cause a memory fault or overwrite data.

Addr should be a multiple of the system page size (0x800), since the limit is modified in page-size increments. Otherwise, it is rounded down to the next page boundary.

Ugrow is automatically called when necessary by the library routine *csav*, which is invoked upon procedure entry. A global variable *_endstk* contains the last value of *addr* passed to *ugrow*; *_endstk* is also used by other library routines.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

NAME

ulimit - get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the process's file size limit. The limit is in units of 1024-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the process's file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM]

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

brk(2), write(2).

NAME

umask - set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), mknod(1M), sh(1), chmod(2), creat(2), mknod(2), open(2).

NAME

umount - unmount a file system

SYNOPSIS

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

Umount will fail if one or more of the following are true:

The process's effective user ID is not super-user. [EPERM]

Spec does not exist. [ENXIO]

Spec is not a block special device. [ENOTBLK]

Spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

Spec points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(1M), mount(2).

NAME

uname - get name of current UNIX system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current UNIX system in the structure pointed to by *name*.

Uname uses the structure defined in **<sys/utsname.h>**:

```
struct utsname {
        char    sysname[9];
        char    nodename[9];
        char    release[9];
        char    version[9];
};
extern struct utsname utsname;
```

Uname returns a null-terminated character string naming the current UNIX system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system.

Uname will fail if *name* points to an invalid address. [EFAULT]

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

SEE ALSO

uname(1).

NAME

unlink - remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to be *path*.

The named file is unlinked unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Write permission is denied on the directory containing the link to be removed. [EACCES]

The named file is a directory and the effective user ID of the process is not super-user. [EPERM]

The entry to be unlinked is the mount point for a mounted file system. [EBUSY]

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

Path points outside the process's allocated address space. [EFAULT]

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1), close(2), link(2), open(2).

NAME

ustat - get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes to following elements:

```
daddr_t f_tfree;        /* Total free blocks */
ino_t   f_tinode;       /* Number of free inodes */
char    f_fname[6];     /* Filsys name */
char    f_fpack[6];     /* Filsys pack name */
```

Ustat will fail if one or more of the following are true:

Dev is not the device number of a device containing a mounted file system. [EINVAL]

Buf points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2), fs(5).

NAME

utime - set file access and modification times

SYNOPSIS

#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;

DESCRIPTION

Path points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is **NULL**, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not **NULL**, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct  utimbuf {
        time_t  actime;    /* access time */
        time_t  modtime;   /* modification time */
};
```

Utime will fail if one or more of the following are true:

The named file does not exist. [ENOENT]

A component of the path prefix is not a directory. [ENOTDIR]

Search permission is denied by a component of the path prefix. [EACCES]

The effective user ID is not super-user and not the owner of the file and *times* is not **NULL**. [EPERM]

The effective user ID is not super-user and not the owner of the file and *times* is **NULL** and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

Times is not **NULL** and points outside the process's allocated address space. [EFAULT]

Path points outside the process's allocated address space. [EFAULT]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2).

NAME

wait - wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

DESCRIPTION

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(2)), or until any one of the calling process's child processes stops in a trace mode (see *ptrace*(2)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and pass useful information to the parent. This is accomplished in the following manner:

> If the child process stopped, the high order 8 bits of status will be zero and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit*; see *exit*(2).
>
> If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal*(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see *intro*(2).

Wait will fail and return immediately if one or more of the following are true:

> The calling process has no existing unwaited-for child processes. [ECHILD]
>
> *Stat_loc* points to an illegal address. [EFAULT]

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of -1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), pause(2), signal(2).

WARNING

See *WARNING* in *signal*(2).

NAME
write - write on a file

SYNOPSIS
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION
Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

> *Fildes* is not a valid file descriptor open for writing. [EBADF]
>
> An attempt is made to write to a pipe that is not open for reading by any process. [EPIPE and SIGPIPE signal]
>
> An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2). [EFBIG]
>
> *Buf* points outside the process's allocated address space. [EFAULT]

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO), no partial writes will be permitted. Thus, the write will fail if a write of *nbyte* bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

RETURN VALUE
Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

SEE ALSO
creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

NAME

intro - introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **-lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3M) These functions constitute the math library, *libm*. 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*(3S)). 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. The files in which these libraries are found are given on the appropriate pages.

The descriptions of some functions refer to **NULL**. This is the value that is obtained by casting **0** into a character pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it, for example, to indicate an error. **NULL** is defined in **<stdio.h>** as **0**; the user can include his own definition if he is not using **<stdio.h>**.

FILES

/lib/libc.a
/lib/libm.a

NOTES

Plexus does not provide *fptrap*(3X), which is specific to non-Plexus hardware and not generally supported in SYSTEM III. Plexus adds *curses* and *termlib*.

SEE ALSO

ar(1), cc(1), f77(1), ld(1), nm(1), intro(2), stdio(3S).

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.

NAME

a64l, l64a - convert between long and base-64 ASCII

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

These routines are used to maintain numbers stored in *base-64* ASCII. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are **.** for 0, / for 1, **0** through **9** for 2-11, **A** through **Z** for 12-37, and **a** through **z** for 38-63.

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. *L64a* takes a **long** argument and returns a pointer to the corresponding base-64 representation.

BUGS

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort - generate an IOT fault

SYNOPSIS

abort ( )

DESCRIPTION

Abort causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored.

SEE ALSO

adb(1), exit(2), signal(2).

DIAGNOSTICS

Usually "abort - core dumped" from the shell.

NAME

abs - integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

SEE ALSO

fabs(3M).

BUGS

You get what the hardware gives on the largest negative integer.

NAME

assert - program verification

SYNOPSIS

#include <assert.h>

assert (expression);

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false, it prints "Assertion failed: file *xyz*, line *nnn*" on the standard error file and exits. *Xyz* is the source file and *nnn* the source line number of the *assert* statement. Compiling with the preprocessor option **-DNDEBUG** (see *cc*(1)) will cause *assert* to be ignored.

NAME

atof, atoi, atol - convert ASCII to numbers

SYNOPSIS

double atof (nptr)
char *nptr;

int 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(3S).

BUGS

There are no provisions for overflow.

NAME

j0, j1, jn, y0, y1, yn - bessel functions

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x);
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

DIAGNOSTICS

Negative arguments cause *y0*, *y1*, and *yn* to return a huge negative value.

NAME

bsearch - binary search

SYNOPSIS

char *bsearch (key, base, nel, width, compar)
char *key;
char *base;
int nel, width;
int (*compar)();

DESCRIPTION

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating the location at which a datum may be found. The table must be previously sorted in increasing order. The first argument is a pointer to the datum to be located in the table. The second argument is a pointer to the base of the table. The third is the number of elements in the table. The fourth is the width of an element in bytes. The last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

DIAGNOSTICS

Zero is returned if the key can not be found in the table.

SEE ALSO

lsearch(3C), qsort(3C).

NAME

toupper, tolower, toascii - character translation

SYNOPSIS

#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

DESCRIPTION

Toupper and *tolower* have as domain the range of *getc*: the integers from -1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted domains and are faster. *_toupper* requires a lower-case letter as its argument; its result is the corresponding upper-case letter. *_tolower* requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause garbage results.

Toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype(3C).

NAME

crypt, setkey, encrypt - DES encryption

SYNOPSIS

char *crypt (key, salt)
char *key, *salt;

setkey (key)
char *key;

encrypt (block, edflag)
char *block;
int edflag;

DESCRIPTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard (DES), 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./]**; this *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 *setkey* and *encrypt* 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

login(1), passwd(1), getpass(3C), passwd(5).

BUGS

The return value points to static data that are overwritten by each call.

Passwords encrypted under V7 use the German Enigma method, which is incompatible with DES.

NAME

ctermid - generate file name for terminal

SYNOPSIS

#include <stdio.h>

char *ctermid(s)
char *s;

DESCRIPTION

Ctermid generates a string that refers to the controlling terminal for the current process when used as a file name.

If (int)*s* is zero, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. If (int)*s* is non-zero, then *s* is assumed to point to a character array of at least **L_ctermid** elements; the string is placed in this array and the value of *s* is returned. The manifest constant **L_ctermid** is defined in **<stdio.h>**.

NOTES

The difference between *ctermid* and *ttyname*(3C) is that *ttyname* must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a magic string (**/dev/tty**) that will refer to the terminal if used as a file name. Thus *ttyname* is useless unless the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, tzset - convert date and time to ASCII

SYNOPSIS

```
#include <time.h>

char cbuf[26];
int dmsize[12];
long timezone;
char *tzname[];
int daylight;
struct {
     int daylb;
     int dayle;
} daytab[];

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

tzset ( )
```

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 the UNIX system 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 non-zero if daylight saving time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named **TZ** is present, *asctime* uses the contents of the variable to override the default time zone. The value of **TZ** must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting **TZ** are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

char *tzname[2] = {"EST", "EDT"};

are set from the environment variable. The function *tzset* sets the external variables from **TZ**; it is called by *asctime* and may also be called explicitly by the user.

SEE ALSO

time(2), getenv(3C), environ(7).

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii - character classification

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int 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*(3S)).

isalpha	*c* is a letter
isupper	*c* is an upper case letter
islower	*c* is a lower case letter
isdigit	*c* is a digit [0-9]
isxdigit	*c* is a hexidecimal digit [0-9], [A-F] or [a-f]
isalnum	*c* is an alphanumeric
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde)
isgraph	*c* is a printing character, like *isprint* except false for space
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 -ltermlib** [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.

SEE ALSO

termcap (5), stty (2), setenv (3), setenv (3).

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
cbreak()	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*
delwin(win)	delete *win*
echo()	set echo mode
erase()	erase *stdscr*
getch()	get a char through *stdscr*
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
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
nocbreak()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unser 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*
restty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf from *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
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*
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) from *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*

NAME

cuserid - character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

Cuserid generates a character representation of the login name of the owner of the current process. If (int)*s* is zero, this representation is generated in an internal static area, the address of which is returned. If (int)*s* is non-zero, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The manifest constant **L_cuserid** is defined in **<stdio.h>**.

DIAGNOSTICS

If the login name cannot be found, *cuserid* returns **NULL**; if *s* is non-zero in this case, **\0** will be placed at **s*.

SEE ALSO

getlogin(3C), getpwuid(3C).

BUGS

Cuserid uses *getpwnam*(3C); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.
The name *cuserid* is rather a misnomer.

NAME

ecvt, fcvt - 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(3S).

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 the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**-p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by "sbrk(0)" (see *brk*(2)).

These symbols are accessible from assembly language if it is remembered that they should be prefixed by _.

SEE ALSO

brk(2), malloc(3C).

NAME

exp, log, pow, sqrt - exponential, logarithm, power, square root functions

SYNOPSIS

#include <math.h>

double exp (x)
double x;

double log (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*.

Pow returns x^y.

Sqrt returns the square root of *x*.

SEE ALSO

intro(2), hypot(3M), sinh(3M).

DIAGNOSTICS

Exp and *pow* return a huge value when the correct value would overflow. A truly outrageous argument may also result in *errno* being set to **ERANGE**.

Log returns a huge negative value and sets *errno* to **EDOM** when *x* is non-positive.

Pow returns a huge negative value and sets *errno* to **EDOM** when *x* is non-positive and *y* is not an integer, or when *x* and *y* are both zero.

Sqrt returns 0 and sets *errno* to **EDOM** when *x* is negative.

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (stream)
FILE *stream;

int 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.

These functions return 0 for success, and **EOF** if any errors were detected.

SEE ALSO

close(2), fopen(3S), setbuf(3S).

NAME

ferror, feof, clearerr, fileno - stream status inquiries

SYNOPSIS

#include <stdio.h>

int feof (stream)
FILE *stream;

int 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 error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

Clearerr resets the error indication on the named *stream*.

Fileno returns the integer file descriptor associated with the *stream*, see *open*(2).

Feof, *ferror*, and *fileno* are implemented as macros; they cannot be re-declared.

SEE ALSO

open(2), fopen(3S).

NAME

floor, fabs, ceil, fmod - absolute value, floor, ceiling, remainder functions

SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

DESCRIPTION

Fabs returns $|x|$.

Floor returns the largest integer (as a double precision number) not greater than *x*.

Ceil returns the smallest integer not less than *x*.

Fmod returns the number *f* such that $x = iy + f$, for some integer *i*, and $0 \le f < y$.

SEE ALSO

abs(3C).

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (file-name, type)
char *file-name, *type;

FILE *freopen (file-name, type, stream)
char *file-name, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Fopen opens the file named by *file-name* 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
"r+"	open for update (reading and writing)
"w+"	create for update
"a+"	append; open or create for update at end of file

Freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed, regardless of whether the open ultimately succeeds.

Freopen is typically used to attach the preopened constant names **stdin**, **stdout**, and **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.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which encounters end of file.

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

Fopen and *freopen* return the pointer **NULL** if *file-name* cannot be accessed.

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

#include <stdio.h>

int fread ((char *) ptr, sizeof (*ptr), nitems, stream)
FILE *stream;

int fwrite ((char *) 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.

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(3S), getc(3S), putc(3S), gets(3S), puts(3S), printf(3S), scanf(3S).

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>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell (stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc*(3S).

After *fseek* or *rewind*, the next operation on an update file may be either input or output.

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. The offset is measured in bytes.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

Fseek returns non-zero for improper seeks, otherwise zero.

NAME

gamma - log gamma function

SYNOPSIS

#include <math.h>
extern int signgam;

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 fragment might be used to calculate Γ:

```
y = gamma (x);
if (y > 88.0)
        error ();
y = exp (y) * signgam;
```

DIAGNOSTICS

For negative integer arguments, a huge value is returned, and *errno* is set to **EDOM**.

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 therefore be used as an argument. *Fgetc* runs more slowly than *getc*, but takes less space per invocation.

Getw returns the next word from the named input *stream*. It returns the constant **EOF** upon end of file or error, but since that is a valid integer value, *feof* and *ferror*(3S) should be used to check the success of *getw*. *Getw* assumes no special alignment in the file.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S).

DIAGNOSTICS

These functions return the integer constant **EOF** at end of file or upon read error.

A stop with message "Reading bad file" means that an attempt has been made to read from a stream that has not been opened for reading by *fopen*.

BUGS

Getc and its variant *getchar* return **EOF** on end of file; this is wiser than, but incompatible with, the older *getchar*(3S).
Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. 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*(7)) for a string of the form *name*=*value* and returns *value* if such a string is present, otherwise 0 (**NULL**).

SEE ALSO

environ(7).

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;

int setgrent ( );

int 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 reads the next line of the file, so successive calls may be used to search the entire file. *Getgrgid* and *getgrnam* search from the beginning of the file until a matching *gid* or *name* is found, or EOF is encountered.

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(3C), getpwent(3C), 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 user ID 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 call *cuserid*, or to call *getlogin* and if it fails, to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(5).

DIAGNOSTICS

Returns **NULL** if name not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getopt - get option letter from argv

SYNOPSIS

int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;
extern char *optarg;
extern int optind;

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **EOF**. The special option -- may be used to delimit the end of the options; **EOF** will be returned, and -- will be skipped.

DIAGNOSTICS

Getopt prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter not included in *optstring*.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc();
                        break;
                case 'f':
                        ifile = optarg;
```

```
                        break;
                case 'o':
                        ofile = optarg;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (2);
        }
        for( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
        :
}
```

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(3C).

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw - get name from UID

SYNOPSIS

getpw (uid, buf)
int uid;
char *buf;

DESCRIPTION

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), 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 returns a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

```
struct  passwd {
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

The *pw_comment* field is unused; the others have meanings described in *passwd*(5).

Getpwent reads the next line in the file, so successive calls can be used to search the entire file. *Getpwuid* and *getpwnam* search from the beginning of the file until a matching *uid* or *name* is found, or **EOF** is encountered.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(5).

DIAGNOSTICS

Null pointer (0) returned on **EOF** or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;

DESCRIPTION

Gets reads a string into *s* from the standard input stream **stdin**. The string is terminated by a new-line character, which is replaced in *s* by a null character. *Gets* returns its argument.

Fgets reads *n*-1 characters, or up to a new-line character (which is retained), 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

ferror(3S), fopen(3S), fread(3S), getc(3S), puts(3S), scanf(3S).

DIAGNOSTICS

Gets and *fgets* return the constant pointer **NULL** upon end-of-file or error.

NOTE

Gets deletes the new-line ending its input, but *fgets* keeps it.

NAME

hypot - Euclidean distance

SYNOPSIS

```
#include <math.h>

double hypot (x, y)
double x, y;
```

DESCRIPTION

Hypot returns

sqrt(x*x + y*y),

taking precautions against unwarranted overflows.

SEE ALSO

sqrt(3M).

NAME

l3tol, ltol3 - convert between 3-byte integers and long integers

SYNOPSIS

l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;

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 block numbers are three bytes long.

SEE ALSO

fs(5).

NAME

logname - login name of user

SYNOPSIS

char *logname();

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

This routine is kept in **/lib/libPW.a.**

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(5), environ(7).

NAME

lsearch - linear search and update

SYNOPSIS

char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
int *nelp;
int width;
int (*compar)();

DESCRIPTION

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm Q. It returns a pointer into a table indicating the location at which a datum may be found. If the item does not occur, it is added at the end of the table. The first argument is a pointer to the datum to be located in the table. The second argument is a pointer to the base of the table. The third is the address of an integer containing the number of items in the table. It is incremented if the item is added to the table. The fourth is the width of an element in bytes. The last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return zero if the items are equal and non-zero otherwise.

BUGS

Unpredictable events can occur if there is not enough room in the table to add a new item.

SEE ALSO

bsearch(3C), qsort(3C).

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 elem, 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 *brk*(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. When *realloc* returns 0, the block pointed to by *ptr* may be destroyed.

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 **Xs**, which will be replaced with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2).

BUGS

It is possible to run out of letters.

NAME

monitor - prepare execution profile

SYNOPSIS

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

DESCRIPTION

An executable program created by **cc -p** automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **-p** of *cc*(1) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext();
...
monitor(2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text, see *end*(3C).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor(0);
```

prof(1) can then be used to examine the results.

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2).

NAME

nlist - get entries from name list

SYNOPSIS

#include <a.out.h>
nlist (file-name, nl)
char *file-name;
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 a discussion of the symbol table structure.

This subroutine is useful for examining the system name list kept in the file **/sys3**. In this way programs can obtain system addresses that are up to date.

NOTES

The system name is **/sys3**, not **/unix**.

SEE ALSO

a.out(5).

DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

perror, sys_errlist, sys_nerr, errno - system error messages

SYNOPSIS

perror (s)
char *s;

int sys_nerr;
char *sys_errlist[ ];

int errno;

DESCRIPTION

Perror produces a short error message on the standard error, describing the last error encountered during a system call from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. To be of most use, the argument string should be the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2).

NAME

plot - graphics interface subroutines

SYNOPSIS

openpl ()

erase ()

label (s)
char *s;

line (x1, y1, x2, y2)

circle (x, y, r)

arc (x, y, x0, y0, x1,

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 terminated by nulls and do not contain new-lines.

The library files listed below provide several flavors of these routines.

FILES

/usr/lib/libplot.a produces output for *tplot*(1G) filters
/usr/lib/lib300.a for DASI 300
/usr/lib/lib300s.a for DASI 300s
/usr/lib/lib450.a for DASI 450
/usr/lib/lib4014.a for Tektronix 4014

SEE ALSO

graph(1G), tplot(1G), plot(5).

NAME

popen, pclose - initiate I/O to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int 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. *Popen* 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), wait(2), fclose(3S), fopen(3S), system(3S).

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns -1 if *stream* is not associated with a "*popen*ed" command.

BUGS

Only one stream opened by *popen* can be in use at once.

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*(3S).

NAME

printf, fprintf, sprintf - output formatters

SYNOPSIS

#include <stdio.h>

int printf (format [, arg] ... **)**
char *format;

int fprintf (stream, format [, arg] ... **)**
FILE *stream;
char *format;

int 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", followed by the null character (**\0**) in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (see below) has been given) to the field width;
>
> A *precision* that gives the minimum number of digits to appear for the **d, o, u, x,** or **X** conversions, the number of digits to appear after the decimal point for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string: a null digit string is treated as zero.
>
> An optional **l** specifying that a following **d, o, u, x,** or **X** conversion character applies to a long integer *arg*.
>
> A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

- **-** The result of the conversion will be left-justified within the field.
- **+** The result of a signed conversion will always begin with a sign (+ or -).
- **blank** If the first character of a signed conversion is not a sign, a blank will be prepended to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (**X**) conversion, a non-zero result will have **0x** (**0X**) prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (**x** and **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string (unless the conversion is **o**, **x**, or **X** *and* the # flag is present).

f The float or double *arg* is converted to decimal notation in the style "[-]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, 6 digits are output; if the precision is explicitly 0, no decimal point appears.

e,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 of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains exactly two digits.

g,G The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (**\0**) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

% Print a **%**; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putchar* had been called (see *putc*(3S)).

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, %.2d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

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)
int w;
FILE *stream;

DESCRIPTION

Putc appends the character *c* to the named output *stream*. It returns the character written.

Putchar(*c*) is defined as *putc*(*c, stdout*).

Fputc behaves like *putc*, but is a genuine function rather than a macro; it may therefore be used as an argument. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw appends the word (i.e., integer) *w* to the output *stream*. *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*(3S). The standard stream *stderr* is by default unbuffered unconditionally, but use of *freopen*(3S) will cause it to become unbuffered; *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. See also *fflush*(3S).

SEE ALSO

ferror(3S), fopen(3S), fwrite(3S), getc(3S), printf(3S), puts(3S).

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, *ferror*(3S) should be used to detect *putw* errors.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++);** doesn't work sensibly.

NAME

putpwent - write password file entry

SYNOPSIS

#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;

DESCRIPTION

Putpwent is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid*(3C) or *getpwnam*(3C)), *putpwuid* writes a line on the stream *f* which matches the format of **/etc/passwd**.

DIAGNOSTICS

Putpwent returns non-zero if an error was detected during its operation, otherwise zero.

NAME

puts, fputs - put a string on a stream

SYNOPSIS

#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;

DESCRIPTION

Puts copies the null-terminated string *s* to the standard output stream *stdout* and appends a new-line character.

Fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminating null character.

DIAGNOSTICS

Both routines return **EOF** on error.

SEE ALSO

ferror(3S), fopen(3S), fwrite(3S), gets(3S), printf(3S), putc(3S).

NOTES

Puts appends a new-line, *fputs* does not.

NAME

qsort - quicker sort

SYNOPSIS

qsort (base, nel, width, compar)
char *base;
int nel, width;
int (*compar)( );

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), strcmp(3C).

NAME

rand, srand - random number generator

SYNOPSIS

srand (seed)
unsigned 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^{15}-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

regex, regcmp - regular expression compile/execute

SYNOPSIS

char *regcmp(string1[,string2, ...],0);
char *string1, *string2, ...;

char *regex(re,subject[,ret0, ...]);
char *re, *subject, *ret0, ...;

DESCRIPTION

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A zero return from *regcmp* indicates an incorrect argument. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns zero on failure or a pointer to the next unmatched character on success. A global character pointer *_loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1) however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[]*.^ These symbols retain their current meaning.

$ Matches the end of the string, **\n** matches the new-line.

- Within brackets the minus means *through*. For example, **[a-z]** is equivalent to **[abcd...xyz]**. The **-** can appear as itself only if used as the last or first character. For example, the character class expression **[]-]** matches the characters **]** and **-**.

+ A regular expression followed by + means *one or more times*. For example, **[0-9]+** is equivalent to **[0-9][0-9]*.**

{m} {m,} {m,u}
Integer values enclosed in **{ }** indicate the number of times the preceding regular expression is to be applied. *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(...)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At present, at most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, e.g. *, +, **{ }**, can work on a single character or a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
    ...
newcursor = regex((ptr=regcmp("^\n",0)),cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
```

```
char *newcursor, *name;
        ...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0",0);
newcursor = regex(name,"123Testing321",ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        ...
newcursor = regex(name,string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), free(3C), malloc(3C).

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3C) re-uses the same vector saving time and space:

```
/* user's program */
        ...
malloc(n) {
static int rebuf[256];
        return &rebuf;
}
```

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 new-lines, which cause input to be read up to the next non-white-space character.
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 an 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 automatically. The input field is terminated by a space character or a new-line.

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, use **%1s**. If a field width is given, the corresponding argument should refer to a character array; 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 an **e**, followed by an optionally signed integer.

[indicates a string that is 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 a circumflex (^), the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a ^, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o**, and **x** may be capitalized and/or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e** and **f** may be capitalized and/or preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The character **h** will, some time in the future, indicate **short** data items.

Scanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

EXAMPLES

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**, to *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* (see *getc*(3S)) will return **a**.

SEE ALSO

atof(3C), getc(3S), printf(3S).

NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

DIAGNOSTICS

These 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*(3C) upon the first *getc* or *putc*(3S) on the file, except that output streams directed to terminals, and the standard error stream *stderr* are normally not buffered.

A common source of error is allocation of buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S).

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

#include <setjmp.h>

int 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 corresponding call to *setjmp*, which must not itself have returned in the interim. *Longjmp* cannot return the value 0. If *longjmp* is invoked with a second argument of 0, it will return 1. All accessible data have values as of the time *longjmp* was called.

SEE ALSO

signal(2).

NAME

sinh, cosh, tanh - hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh (x)
double 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

unsigned 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 less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*; if the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the *sleep* routine returns, but if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

ssignal, gsignal - software signals

SYNOPSIS

#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;

DESCRIPTION

Ssignal and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable the user to indicate the disposition of error conditions, and is also made available to the user for his own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. An *action* for a software signal is *established* by a call to *ssignal*, and a software signal is *raised* by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio - standard buffered input/output package

SYNOPSIS

#include <stdio.h>
FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* 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*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are 3 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 the null stream.

An integer constant EOF (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that "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*, and *fileno*.

SEE ALSO

open(2), close(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpnam(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - string operations

SYNOPSIS

```
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

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*.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or **NULL** if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or **NULL** if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a **NULL** character into *s1* immediately following the returned token. Subsequent calls with zero for the first argument, will work through the string *s1* in this way until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a **NULL** is returned.

BUGS

Strcmp uses native character comparison, which is signed on the Z8000, and unsigned on other machines.

All string movement is performed character by character starting at the left. Thus overlapping moves toward the left will work as expected, but overlapping moves to the right may yield surprises.

NAME

swab - swap bytes

SYNOPSIS

swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. *Nbytes* should be even.

NAME

system - issue a shell command

SYNOPSIS

#include <stdio.h>

int 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

sh(1), exec(2).

DIAGNOSTICS

System stops if it can't execute *sh*(1).

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

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.

Tgetent extracts the entry for a 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 looks 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 help debug new terminal descriptions or 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 *id* 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 the capability *id*, placing it in the buffer at *area*, and 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 that call *tgoto* should 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 that 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 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/libtermlib.a termcap library
/etc/termcap data base

NOTES

These routines are based on those from the University of California at Berkeley.

SEE ALSO

ex(1), termcap(5).

NAME

tmpfile - create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

DESCRIPTION

Tmpfile creates a temporary file and returns a corresponding **FILE** pointer. Arrangements are made so that the file will automatically be deleted when the process using it terminates. The file is opened for update.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S).

NAME

tmpnam - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;
```

DESCRIPTION

Tmpnam generates a file name that can safely be used for a temporary file. If (int)*s* is zero, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If (int)*s* is nonzero, *s* is assumed to be the address of an array of at least **L_tmpnam** bytes; *tmpnam* places its result in that array and returns *s* as its value.

Tmpnam generates a different file name each time it is called.

Files created using *tmpnam* and either *fopen* or *creat* are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C).

BUGS

If called more than 17,576 times in a single process, *tmpnam* will start recycling previously used names.
Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam* or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

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 (y, x)
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 *y*/*x* in the range $-\pi$ to π.

DIAGNOSTICS

Arguments of magnitude greater than 1 cause *asin* and *acos* to return value 0.

NAME

ttyname, isatty, ttyslot - find name of a terminal

SYNOPSIS

char *ttyname (fildes)

int 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 slot in **/etc/utmp** corresponding to the current user.

FILES

/dev/*
/etc/utmp

DIAGNOSTICS

Ttyname returns a null pointer (0) if *fildes* does not describe a terminal device in directory **/dev**.

Ttyslot returns -1 if **/etc/utmp** 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>

int ungetc (c, stream)
char c;
FILE *stream;

DESCRIPTION

Ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *Ungetc* returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push **EOF** are rejected.

Fseek(3S) erases all memory of pushed back characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S).

DIAGNOSTICS

Ungetc returns **EOF** if it can't push a character back.

NAME

intro - introduction to special files

DESCRIPTION

This section describes various special files that refer to specific Plexus peripherals and UNIX device drivers. The names of the entries are generally derived from Plexus names for the hardware, as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding UNIX device driver are discussed where applicable.

NOTES

Plexus does not support some devices because of hardware differences between DEC and Plexus machines. The following devices are not supported: *cat*, *dj*, *dmc*, *dn*, *dqs*, *du*, *dz*, *hp*, *hs*, *ht*, *kl*, *kmc*, *pcl*, *rf*, *rk*, *rl*, *rp*, *tm*, and *vp*. Plexus adds the following: *dk*, *icp*, *is*, *mt*, *pd*, *pp*, *pt*, and *rm*.

BUGS

The names of the entries *generally* refer to Plexus hardware names, but in certain cases these names are arbitrary for various historical reasons.

NAME

dk - pseudo disk driver

DESCRIPTION

Dk is the "generic" disk device. It accesses whatever disk you have; it tries IMSC disks first. If you have both IMSC and iSBC disks, you must use the special file *is* to access the iSBC disk.

FILES

/dev/dk?

NOTES

This is a Plexus feature. It is not part of stock SYSTEM III.

NAME
err - error-logging interface

DESCRIPTION
Minor device 0 of the *err* driver is the interface between a process and the system's error-record collection routines. The driver may be opened only for reading by a single process with super-user permissions. Each read causes an entire error record to be retrieved; the record is truncated if the read request is for less than the record's length.

FILES
/dev/error special file

SEE ALSO
errdemon(1M).

NAME

icp - Intelligent Communications Processor

DESCRIPTION

The *icp* is a special device that allows access to the memory of the Intelligent Communications Processor (ICP). Reading from the device resets the ICP. Writing to the device overwrites the memory.

FILES

/dev/ic[0-4]

BUGS

Reading from the ICP resets it and kills all terminals actively using it.

SEE ALSO

dnld(1m)

NAME

imsp - Intelligent Mass Storage Processor

DESCRIPTION

The *imsp* is a special device that allows access to the memory of the Intelligent Mass Storage processor (IMSP). Reading from the device returns data from the IMSP's local memory. Writing to the device overwrites the IMSP's local memory.

FILES

/dev/im[0-3]

WARNING

Writing to the IMSP can cause it to hang. This may crash UNIX and destroy file systems.

NAME

is - iSBC disk controller

DESCRIPTION

The iSBC disk controller and associated driver code access up to 4 disks. Each disk is subdivided into 16 logical volumes. By convention, /dev/dk[0-15] refer to the logical volumes of disk 0, /dev/dk[16-31] refer to the logical volumes of disk 1, and so on.

The origin and size of the 16 logical volumes on a disk are:

Volume	Starting Block (1024 byte)	Length (in 1024 byte blocks) (˜ refers to end of disk)
0	0	˜
1	0	20000
2	20000	˜
3	30000	˜
4	40000	˜
5	50000	˜
6	60000	˜
7	70000	˜
8	80000	˜
9	90000	˜
10	100000	˜
11	110000	˜
12	120000	˜
13	130000	˜
14	140000	˜
15	150000	˜

The *dk* 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 *is* files begin with *rdk* and end with a number that selects the same logical disk volume as the corresponding *dk* file.

In raw I/O the buffer must begin on a word boundary.

FILES

/dev/dk?

BUGS

In raw I/O *read* and *write*(2) truncate file offsets to 1024-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 1024-byte multiples.

NAME

lp - line printer

DESCRIPTION

The line printer is a special file to which the line printer daemon, *lpd*, prints output. It may be a serial port, *ttyX*, or a parallel port, *ppX*.

FILES

/dev/lp

SEE ALSO

lpr(1), tty(4) pp(4).

NAME

mem, kmem - core memory
mbiomem, mbmem - Multibus memory
liomem - local I/O device memory

DESCRIPTION

Mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system.

Byte addresses in *mem* are interpreted as 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.

Mbiomem is a special file that is an image of the Multibus I/O address space.

Mbmem is a special file that is an image of the Multibus memory address space.

Liomem is a special file that ia an image of the local I/O device address space. This can be used, for example, to reference the clock chip or the SIO chip.

FILES

/dev/mem
/dev/kmem
/dev/mbiomem
/dev/mbmem
/dev/liomem

NAME

mt - pseudo tape driver

DESCRIPTION

Mt is the "generic" tape device. It accesses whatever tape you have -- either 9-track or cartridge. If you have both 9-track and cartridge tapes, *mt* accesses the 9-track, and you may use the special file *pt* to access the cartridge tape or else omit the device specification entirely.

FILES

/dev/mt?

NOTES

This is a Plexus feature. It is not part of stock SYSTEM III.

NAME

null - the null file

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

mt - pseudo tape driver

DESCRIPTION

Mt is the "generic" tape device. It accesses whatever tape you have -- either 9-track or cartridge. If you have both 9-track and cartridge tapes, *pt* accesses the 9-track, and you may use the special file *mt* to access the cartridge tape or else omit the device specification entirely.

FILES

/dev/mt?

NOTES

This is a Plexus feature. It is not part of stock SYSTEM III.

NAME

pd - IMSP disk controller

DESCRIPTION

The IMSP disk/tape controller and associated driver code access up to 4 disks. Each disk is subdivided into 16 logical volumes. By convention, /dev/dk[0-15] refer to the logical volumes of physical disk 0, /dev/dk[16-31] refer to the logical volumes of physical disk 1, and so on.

The origin and size of the 16 logical volumes on a disk are as follows. '~' refers to the end of the physical disk. Length is given in 1024 byte blocks.

Volume	Starting Block	Length	
0	0	~	
1	0	20000	(default swap area is 18000-20000)
2	20000	~	
3	30000	~	
4	40000	~	
5	50000	~	
6	60000	~	
7	70000	~	
8	80000	~	
9	90000	~	
10	100000	~	
11	110000	~	
12	120000	~	
13	130000	~	
14	140000	~	
15	150000	~	

The *dk* 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 pd files begin with *rdk* and end with a number that selects the same logical disk volume as the corresponding *dk* file.

In raw I/O the buffer must begin on a word boundary.

FILES

/dev/dkx

NOTES

This is a Plexus device, not part of standard SYSTEM III.

DIAGNOSTICS

The IMSP controller may produce the following error messages:

0x0201 Reserved for controller busy

0x0301 Command undefined

0x0401 Command cannot be done

0x0501 Bad CAB parameters

0x0f01 Firmware bug encountered

0x0601 Internal command interrupts

0x0701	Parity error occurred
0x0801	PROM checksum error
0x1103	Disk protected from writing
0x1203	Disk not ready
0x1303	Disk drive fault indicated
0x1403	Disk failed to select
0x1503	Disk operation timeout error
0x1603	Disk failed in formatting
0x1703	Disk seek error
0x1803	Disk ECC error in id field
0x1903	Disk ECC error in data field
0x1b03	Disk limits not defined
0x1c03	Disk unable to locate track

NAME

pp - parallel port interface

DESCRIPTION

The parallel port interface enables access to the parallel port on the Intelligent Communications Processor (ICP). Each ICP has one parallel port interface. The parallel port interface is a write-only device. It is also a raw device, i.e., the operating system does no processing of data written to it.

Pp has no *stty*-like features. If your printer does not handle tabs and new-line characters, you need to write a filter to use this device.

FILES

/dev/pp[0-3]

SEE ALSO

lp(4), tty(4), icp(4)

NAME

prf - operating system profiler

DESCRIPTION

The file **prf** provides access to activity information in the operating system. Writing the file loads the measurement facility with text addresses to be monitored. Reading the file returns these addresses and a set of counters indicative of activity between adjacent text addresses.

The recording mechanism is driven by the system clock and samples the program counter at line frequency. Samples that catch the operating system are matched against the stored text addresses and increment corresponding counters for later processing.

The file **prf** is a pseudo-device with no associated hardware.

FILES

/dev/prf

SEE ALSO

config(1M), profiler(1M).

NAME

pt - IMSP cartridge controller

DESCRIPTION

The IMSP disk/tape controller and associated driver code allow access to a cartridge tape. The cartridge can be accessed only in raw mode (i.e., as a character device), and can be rewound or left at the current position. These options are available based on the minor device number of the special file used to access it. If the cartridge is not to be rewound, it is positioned after the filemark at the end of the current file.

If the 04 bit is on in the minor device number, the cartridge is not rewound when closed.

By convention, the files **/dev/rmt0** and **/dev/nrmt0** are used to access the cartridge in raw mode. Accessing **/dev/rmt0** rewinds the cartridge when this special file is closed. Accessing **/dev/nrmt0** does not rewind the cartridge when the file is closed. Each *read* or *write* call reads or writes the next record on the cartridge. All records on a cartridge are 512 bytes long and all reads and writes must be in multiples of 512 bytes. An error is returned otherwise. The I/O buffer used in the *read*(2) or *write*(2) system call should begin on a word boundary and the count should be even. Seeks are ignored. A zero byte count is returned when a file mark is read, but another read will fetch the first record of the new file.

The cartridge drive can be accessed in high speed mode. However, this mode is effectively limited to skipping forward over files on the cartridge and to I/O between the cartridge and a disk attached to the same IMSP controller. High speed mode is accessed via *ioctl*(2) system calls. The arguments to the *ioctl* are:

fildes File descriptor returned from an *open*(2) of the special tape file **/dev/rmt0or /dev/nrmt0**.

request A special command for the cartridge drive. These commands are defined in **/usr/include/sys/imsc.h** and some are described below.

arg A pointer to a structure of the type "ptcmd" as defined in **/usr/include/sys/imsc.h.**

Some of the members of **ptcmd** are:

dknum Major/minor device number of the IMSP disk being read or written to (if applicable) as returned by *stat*(2) system call (st_rdev).

blkno Starting sector number on logical disk to be read/written. Sectors on disk are 512 bytes long and numbered starting at 0. Note sector addresses are relative to the logical, not the physical disk.

blkcnt The number of 512-byte records to be read from or written to cartridge.

Some of the more useful *ioctl* requests for the cartridge as defined in **/usr/include/sys/imsc.h** are:

C_IRECALL Read from cartridge and write to disk. The cartridge and disk must be on same IMSP controller. The system returns in **ptcmd.blkcnt** the number of 512-byte records not read. This is zero if the system reads all the records requested.

C_ISAVE Read from disk and write to tape. The cartridge and disk must be on same IMSP controller. The system returns in **ptcmd.blkcnt** the number of 512-byte records not read. This is zero if the system reads all the record images (sectors) requested.

C_IWEOF Write EOF mark on cartridge.

C_IREW Rewinds the cartridge.

C_IMOVE Position to file **blkcnt** on cartridge.

Writing multiple files on cartridge should be done all at once, i.e., without rewinding the cartridge. Once a cartridge has been rewound, positioning to the end of a file on the cartridge and then writing to the cartridge may overwrite data. For example, once the cartridge has been rewound, positioning to the end of file 2 and writing to the cartridge may overwrite portions of file 2.

Neither the hardware or the software implement or support an end-of-tape marker on the cartridge.

FILES

/dev/rmt0
/dev/nrmt0

DIAGNOSTICS

The IMSP controller may produce the following error diagnostics:

0x0201	Reserved for controller busy
0x0301	Command undefined
0x0401	Command cannot be done
0x0501	Bad CAB parameters
0x0f01	Firmware bug encountered
0x0601	Internal command interrupts
0x0701	Parity error occurred
0x0801	PROM checksum error
0x1004	End of file reached
0x1304	An exception other than an end-of-file error
0x1504	Tape timeout error
0x1604	Error during recall
0x1704	Error during save
0x1804	Error received while attempting to get status from the tape drive
0x1904	During exception state, a command other than *rstat* was received
0x2004	No tape drive present
0x2104	Timeout during wait recall
0x2204	Timeout during wait save
0x2304	Timeout during stat tape
0x2404	Timeout during stat tape
0x2504	Timeout during command tape
0x2604	Timeout during command tape
0x2704	Timeout during ready tape
0x2804	Tape drive inconsistent at start of tape command
0x1505	Timeout on Host bus request

NAME

rm - Cipher Microstreamer tape drive

DESCRIPTION

The Cipher Microstreamer magnetic tape can be accessed in blocked or raw mode and can be rewound or left at the current position. These options are available based on the minor device number of the special file used to access it. When the special file is closed, the tape can be rewound or not (see below). If the special file was open for writing, two end-of-files are written. If the tape is not to be rewound, it is positioned with the head between the two tapemarks.

If the 04 bit is on in the minor device number, the tape is not rewound when closed.

If the 010 bit is on in the minor device number, the tape is set to high speed mode (100 in/sec). By convention, **/dev/nrrmh0** accesses the tape in high speed mode.

By convention, the file **/dev/mt0** accesses the tape in blocked mode. A tape accessed in block mode consists of a series of 1024-byte records terminated by an end-of-file. As much as it can, 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.

Use **/dev/mt0** to access the tape in a way compatible with ordinary files. However, when foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is more appropriate. By convention, the files **/dev/rmt0** and **/dev/nrmt0** are used to access the tape in raw mode. Accessing **/dev/rmt0** rewinds the tape when **/dev/rmt0** is closed. Accessing **/dev/nrmt0** does not rewind the tape when **/dev/nrmt0** is closed.

Each *read* or *write* call reads or writes the next record on the tape. For writes, 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 number of bytes requested; if the record is longer than the number of bytes requested, an error is returned. On the other hand, if the number of bytes requested is larger than the actual record size, there is a delay of 1-2 seconds between the reading of each record.

In raw tape I/O, the buffer must begin on a word boundary and the count must be even. 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.

The tape drive can be run in high speed mode; however, this is really only usable for fast forward or reverse skipping of file marks. The files used for high speed mode are denoted by an 'h' just before the unit number.

There is an ioctl(2) interface for controlling the tape drive. More information about this can be found in */usr/include/sys/rm.h*.

FILES

/dev/mt0
/dev/rmt0
/dev/nrmt0
/dev/nrrmh0

SEE ALSO

tape(1).

DIAGNOSTICS

The tape controller issues the following codes for unrecoverable errors detected during execution of a command. The code is returned in the Command Status byte, bits 0-4.

Code	Description
00	No unrecoverable error.

NAME

rm – Cipher Microstreamer tape drive

DESCRIPTION

The Cipher Microstreamer magnetic tape can be accessed in blocked or raw mode and can be rewound or left at the current position. These options are available based on the minor device number of the special file used to access it. When the special file is closed, the tape can be rewound or not (see below). If the special file was open for writing, two end-of-files are written. If the tape is not to be rewound, it is positioned with the head between the two tapemarks.

If the 04 bit is on in the minor device number, the tape is not rewound when closed.

If the 010 bit is on in the minor device number, the tape is set to high speed mode (100 in/sec). By convention, /**dev/nrrmh0** accesses the tape in high speed mode.

By convention, the file /**dev/mt0** accesses the tape in blocked mode. A tape accessed in block mode consists of a series of 1024-byte records terminated by an end-of-file. As much as it can, 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.

Use /**dev/mt0** to access the tape in a way compatible with ordinary files. However, when foreign tapes are to be dealt with, and especially when long records are to be read or written, the 'raw' interface is more appropriate. By convention, the files /**dev/rmt0** and /**dev/nrmt0** are used to access the tape in raw mode. Accessing /**dev/rmt0** rewinds the tape when /**dev/rmt0** is closed. Accessing /**dev/nrmt0** does not rewind the tape when /**dev/nrmt0** is closed.

Each *read* or *write* call reads or writes the next record on the tape. For writes, 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 number of bytes requested; if the record is longer than the number of bytes requested, an error is returned. On the other hand, if the number of bytes requested is larger than the actual record size, there is a delay of 1-2 seconds between the reading of each record.

In raw tape I/O, the buffer must begin on a word boundary and the count must be even. 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.

The tape drive can be run in high speed mode; however, this is really only usable for fast forward or reverse skipping of file marks. The files used for high speed mode are denoted by an 'h' just before the unit number.

If you want to write your own program for tape manipulation on the *rm* device, there is an **ioctl**(2) interface for controlling the tape drive. The file */usr/include/sys/rm.h* lists the commands that can be issued. These all begin with "C_" (capital C followed by an underbar). The only **ioctl** request type allowed for this device is RMPOSN ("*rm* position"). The **ioctl** call structure is

```
struct rmcmd_struct {
        unsigned rm_cmd;        /* the command C_<option> */
        unsigned rm_cnt;        /* count, useful for commands such as SRCHEOF */
        unsigned rm_status;     /* physical device status returned */
};
```

The status value is found by adding all the relevant values in the "status fields" portion of *rm.h*. Status is determined by the output status field, which consists of two bytes arranged as follows:

```
 15   14   13   12   11 10   9 8   7     6     5      4      3    2     1
---------------------------------------------------------------------------
|    |    |    |                 |     |     |     |      |     |     |   |
| E  | C  | R  |   ERROR         | FM  | OL  | LP  | EOT  |  R  | FB  | P |
|____|____|____|_________________|_____|_____|_____|______|_____|_____|___|
```

where

Byte 0	Not used
P	(Write Protect) The tape does not have a write enable ring.
FB	(Formatter Busy) The Formatter is busy.
R	(Ready) The selected drive is ready.
EOT	(End of Tape) The EOT marker was detected.
LP	(Load Point) The tape is at load point.
OL	(On Line) The drive is on line.
FM	(Filemark) A filemark was detected on this operation.
E	(Entered) Execution has begun.
C	(Complete) The command has completed successfully.
R	(Retry) At least one Retry was executed .
ERROR	This 5-bit field specifies an error code when a non-recoverable error is encountered. Error codes are listed under DIAGNOSTICS below.

For example, the value "C068" means the tape is online at load point, ready, and previous command has completed.

The following program fragment illustrates the use of **ioctl** to rewind the tape.

```
#include "sys/rm.h"
#include "fcntl.h"
int fildes;          /* file descriptor, returned by open */

fildes = open("/dev/rmt0",O_RDWR);

rmcmd.cmd = C_REW;
rmcmd.cnt = 1;
rmcmd.status = -1;

ioctl(fildes, RMPOSN, &rmcmd);
```

FILES

/dev/mt0
/dev/rmt0
/dev/nrmt0
/dev/nrrmh0
/usr/include/sys/rm.h

SEE ALSO

tape(1), ioctl(2).

DIAGNOSTICS

The tape controller issues the following codes for unrecoverable errors detected during execution of a command. The code is returned in the Command Status byte, bits 8-12.

Code	Description
00	No unrecoverable error.
01	Timed out waiting for expected Data Busy false.
02	Timed out waiting for expected Data Busy false, Formatter Busy false and Ready True.
03	Timed out waiting for expected Ready false.
04	Timed out waiting for expected Ready true.
05	Timed out waiting for expected Data Busy true.
06	A memory time-out occurred during a system memory reference.
07	A blank tape was encountered where data was expected.
08	An error occurred in the micro-diagnostic.
09	An unexpected EOT was encountered during a forward operation, or Load Point during a reverse operation.
0A	A hard or soft error occurred that could not be eliminated by retry.
0B	A read overflow or write overflow occurred. This error indicates that the FIFO was empty when data was requested by the tape during a write, or full when the tape presented a byte during a read.
0C	Not used.
0D	A read parity error occurred on the byte interface between the drive and the controller.
0E	An error was detected during calculation of the checksum on the PROM.
0F	A tape time-out occurred, because the tape drive did not supply an expected read or write strobe. This error occurs when you attempt to read a larger record than was written. It may also occur during a write if the tape is damaged.
10	Tape not ready.
11	A write was attempted on a tape without a write-enable ring.
12	Not used.
13	The diagnostic mode jumper was not installed while attempting to execute a Diagnostic command.
14	An attempt was made to link from a command that does not allow linking.
15	An unexpected filemark was encountered during a tape read.
16	An error in specifying a parameter was detected by the controller. The usual cause is a byte count that is either zero or too large.
17	Not used.
18	An unidentifiable hardware error occurred.
19	A streaming read or write operation was terminated by the operating system or disk.

The tape driver sends the code FFFF to the screen when the block size requested is smaller than the actual block size on the tape.

01	Timed out waiting for expected Data Busy false.
02	Timed out waiting for expected Data Busy false, Formatter Busy false and Ready True.
03	Timed out waiting for expected Ready false.
04	Timed out waiting for expected Ready true.
05	Timed out waiting for expected Data Busy true.
06	A memory time-out occurred during a system memory reference.
07	A blank tape was encountered where data was expected.
08	An error occurred in the micro-diagnostic.
09	An unexpected EOT was encountered during a forward operation, or Load Point during a reverse operation.
0A	A hard or soft error occurred that could not be eliminated by retry.
0B	A read overflow or write overflow occurred. This error indicates that the FIFO was empty when data was requested by the tape during a write, or full when the tape presented a byte during a read.
0C	Not used.
0D	A read parity error occurred on the byte interface between the drive and the controller.
0E	An error was detected during calculation of the checksum on the PROM.
0F	A tape time-out occurred, because the tape drive did not supply an expected read or write strobe. This error occurs when you attempt to read a larger record than was written. It may also occur during a write if the tape is damaged.
10	Tape not ready.
11	A write was attempted on a tape without a write-enable ring.
12	Not used.
13	The diagnostic mode jumper was not installed while attempting to execute a Diagnostic command.
14	An attempt was made to link from a command that does not allow linking.
15	An unexpected filemark was encountered during a tape read.
16	An error in specifying a parameter was detected by the controller. The usual cause is a byte count that is either zero or too large.
17	Not used.
18	An unidentifiable hardware error occurred.
19	A streaming read or write operation was terminated by the operating system or disk.

The tape driver sends the code FFFF to the screen when the block size requested is smaller than the actual block size on the tape.

NAME

st - synchronous terminal interface

DESCRIPTION

The synchronous terminal interface is a pseudo-device driver that enables a UNIX system to communicate with a TELETYPE® Model 40/4 ASCII synchronous terminal. The driver utilizes the Virtual Protocol Machine (VPM) to perform the end-to-end protocol and transmission assurance for the synchronous line.

The user must be familiar with the operation of the Model 40/4 terminal. Screen management functions are completely controlled by the user process; when formating a screen, the user must supply everything from the initial STX (Start-of-Text) character to the ETX (End-of-Text) character.

By convention, **/dev/st0** is the synchronous terminal control channel, while other **/dev/st?** files represent user terminal channels. Communication with the control channel is handled by the *stcntrl* command (see *st*(1M)).

A user process will sleep when trying to open a channel, until a terminal requests service. At that time, a channel will be assigned to that terminal, and it will remain allocated until the user process closes the terminal.

In addition to the synchronous terminal equipment, a KMC11-B microprocessor, and a DMC11-DA synchronous line unit are required.

FILES

/etc/stproto	synchronous terminal prototype script
/dev/kmc?	KMC11-B microprocessor
/dev/vpm?	virtual protocol machine
/dev/st0	synchronous terminal control channel
/dev/st?	synchronous terminal user channels

SEE ALSO

st(1M), kmc(4), trace(4), vpm(4).

NAME

swap - image of the swap area

DESCRIPTION

swap is a block special device that corresponds to the file system containing the swap area (default **/dev/dk1**). Reading from the *swap* device returns data from the swap area.

FILES

/dev/swap

NAME

trace - event-tracing driver

DESCRIPTION

Trace is a special file that allows UNIX kernel drivers to transfer event records to a user program, so that the activity of the driver may be monitored for debugging purposes.

An event record is generated from within a kernel driver by executing the following function:

```
trsave(dev, chno, buf, cnt)
char    dev, chno, *buf, cnt;
```

Dev is the minor device number of the trace driver; *chno* is an integer between 1 and 16, inclusive, identifying the data stream to which the record belongs; *buf* is a buffer containing the bytes that make up a single event record; and *cnt* is the number of bytes in *buf*. Calls to *trsave* will result in data being saved in a *clist* buffer, provided that some user program has opened the trace minor device number *dev* and has activated channel *chno*. Event records prefaced by *chno* and *cnt* are stored in a *clist* queue until a system-defined maximum (TRQMAX) is reached; event records are discarded while the queue is full. The *clist* queue is emptied by a user program reading the trace driver. The trace driver returns an integral number of event records; the read count must, therefore, be at least equal to the size of a record plus two, to allow for the *chno* and *cnt* bytes added to the event record by the *trsave* routine.

The *trace* driver supports *open*, *close*, *read*, and *ioctl* system calls. To activate a channel, *ioctl* is used as follows:

```
#include <ioctl.h>
ioctl(fildes, VPMTRCO, chno)
```

SEE ALSO

vpmstart(1C), vpm(4).

NAME

tty - general terminal interface

DESCRIPTION

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

The file **/dev/tty** is, in each process, a synonym for the control terminal associated with the process group of that process, if any. It is useful for programs or shell sequences 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 the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

As for terminals in general: all of the asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by *getty*(8) and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *control terminal* for that process group. The control terminal plays a special role in handling quit and interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*(2). A process can break this association by changing its process group using *setpgrp*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely full, 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 512 characters. When the input limit is reached, all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, erase and kill processing is normally done. By default, the character **#** erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the character **@** kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a key-stroke basis, independently of any backspacing or tabbing that may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (****). In this case the escape character is not read. The erase and kill characters may be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (Rubout or ASCII DEL) generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see *signal*(2).

QUIT (Control-**|** or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (#) erases the preceding character. It will not erase beyond the start of a line, as delimited by a NL, EOF, or EOL character.

KILL (@) deletes the entire line, as delimited by a NL, EOF, or EOL character.

EOF (Control-d or ASCII EOT) may be used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

EOL (ASCII NUL) is an additional line delimiter, like NL. It is not normally used.

STOP (Control-s or ASCII DC3) can be used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (Control-q or ASCII DC1) is used to resume output which has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can not be changed or escaped.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL may be changed to suit individual tastes. The ERASE, KILL, and EOF characters may be escaped by a preceding \ character, in which case no special function is done.

When the carrier signal from the data-set drops, a *hangup* signal is sent to all processes that have this terminal as the control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If 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.

Several *ioctl*(2) system calls apply to terminal files. The primary calls use the following structure, defined in **<termio.h>**:

```
#define NCC      8
struct  termio {
        unsigned short  c_iflag;     /* input modes */
        unsigned short  c_oflag;     /* output modes */
        unsigned short  c_cflag;     /* control modes */
        unsigned short  c_lflag;     /* local modes */
        char            c_line;      /* line discipline */
        unsigned char   c_cc[NCC];   /* control chars */
};
```

The special control characters are defined by the array *c_cc*. The relative positions and initial values for each function are as follows:

0	INTR	DEL
1	QUIT	FS
2	ERASE	#
3	KILL	@
4	EOF	EOT

5	EOL	NUL
6	reserved	
7	reserved	

The *c_iflag* field describes the basic terminal input control:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.
INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map NL to CR on input.
IGNCR	0000200	Ignore CR.
ICRNL	0000400	Map CR to NL on input.
IUCLC	0001000	Map upper-case to lower-case on input.
IXON	0002000	Enable start/stop output control.
IXANY	0004000	Enable any character to restart output.
IXOFF	0010000	Enable start/stop input control.

See NOTES below for Plexus additions to this list.

If IGNBRK is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise if BRKINT is set, the break condition will generate an interrupt signal and flush both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

If PARMRK is set, a character with a framing or parity error which is not ignored is read as the three character sequence: 0377, 0, X, where X is the data of the character received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid character of 0377 is read as 0377, 0377. If PARMRK is not set, a framing or parity error which is not ignored is read as the character NUL (0).

If INPCK is set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled. This allows output parity generation without input parity errors.

If ISTRIP is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

If INLCR is set, a received NL character is translated into a CR character. If IGNCR is set, a received CR character is ignored (not read). Otherwise if ICRNL is set, a received CR character is translated into a NL character.

If IUCLC is set, a received upper-case alphabetic character is translated into the corresponding lower-case character.

If IXON is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read. If IXANY is set, any input character will restart output that has been suspended. Note that some terminals experience difficulty with IXANY.

If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

The *c_oflag* field specifies the system treatment of output:

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lower case to upper on output.
ONLCR	0000004	Map NL to CR-NL on output.
OCRNL	0000010	Map CR to NL on output.

ONOCR	0000020	No CR output at column 0.
ONLRET	0000040	NL performs CR function.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL, else NUL.
NLDLY	0000400	Select new-line delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:
BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

If OPOST is set, output characters are post-processed as indicated by the remaining flags, otherwise characters are transmitted without change.

If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

If ONLCR is set, the NL character is transmitted as the CR-NL character pair. If OCRNL is set, the CR character is transmitted as the NL character. If ONOCR is set, no CR character is transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the CR character is actually transmitted.

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. If OFILL is set, fill characters will be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character is DEL, otherwise NUL.

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

New-line delay lasts about 0.10 seconds. If ONLRET is set, the carriage-return delays are used instead of the new-line delays. If OFILL is set, two fill characters will be transmitted.

Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds. If OFILL is set, delay type 1 transmits two fill characters, and type 2 four fill characters.

Horizontal-tab delay type 1 is dependent on the current column position. Type 2 is about 0.10 seconds. Type 3 specifies that tabs are to be expanded into spaces. If OFILL is set, two fill characters will be transmitted for any delay.

Backspace delay lasts about 0.05 seconds. If OFILL is set, one fill character will be transmitted.

The actual delays depend on line speed and system load.

The initial output control value is all bits clear.

The *c_cflag* field describes the hardware control of the terminal:

CBAUD	0000017	Baud rate:
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
EXTA	0000016	External A (19200 baud)
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send two stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.

The CBAUD bits specify the baud rate. The zero baud rate, B0, is used to hang up the connection. If B0 is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, impossible speed changes are ignored.

The baud rate for EXTB is determined from switch settings in the hardware. See the *Plexus User's Manual* for details.

The CSIZE bits specify the character size in bits for both transmission and reception. This size does not include the parity bit, if any. If CSTOPB is set, two stop bits are used, otherwise one stop bit. For example, at 110 baud, two stops bits are required.

If PARENB is set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the PARODD flag specifies odd parity if set, otherwise even parity is used.

If CREAD is set, the receiver is enabled. Otherwise no characters will be received.

If HUPCL is set, the line will be disconnected when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

If CLOCAL is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

The *c_lflag* field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo NL after kill character.
ECHONL	0000100	Echo NL.
NOFLSH	0000200	Disable flush after interrupt or quit.

If ISIG is set, each input character is checked against the special control characters INTR and QUIT. If an input character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, no checking is done. Thus these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g. 0377).

If ICANON is set, canonical processing is enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, *read*(2) requests are satisfied directly from the input queue. A *read* will not be satisfied until at least MIN characters have been received or the timeout value TIME has expired. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters respectively. The time value represents tenths of seconds; values for TIME range from 2 to 255. If TIME has the value 0 or 1, no timeout occurs.

If XCASE is set, and if ICANON is set, an upper-case letter is accepted on input by preceding it with a \ character, and is output preceded by a \ character. In this mode, the following escape sequences are generated on output and accepted on input:

for:	*use*:
`	\'
|	\\!
	\\^
{	\\(
}	\\)
\\	\\\\

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

If ECHO is set, characters are echoed as received.

When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which will clear the last character from a CRT screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the NL character will be echoed after the kill character to emphasize that the line will be deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the NL character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (so-called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done. When NOFLSH is set, a *del* (0177) or a ^| will cause a signal to be sent to the process. This process will be terminated. The character has already been placed in the raw queue and will be read with the next **read**.

The initial line-discipline control value is all bits clear.

The primary *ioctl*(2) system calls have the form:

```
ioctl (fildes, command, arg)
struct termio *arg;
```

The commands using this form are:

TCGETA Get the parameters associated with the terminal and store in the *termio* structure referenced by **arg**.

TCSETA Set the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.

TCSETAW Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.

TCSETAF Wait for the output to drain, then flush the input queue and set the new parameters.

Additional *ioctl*(2) calls have the form:

```
ioctl (fildes, command, arg)
int arg;
```

The commands using this form are:

TCSBRK Wait for the output to drain. If *arg* is 0, then send a break (zero bits for 0.25 seconds).

TCXONC Start/stop control. If *arg* is 0, suspend output; if 1, restart suspended output.

TCFLSH If *arg* is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

FILES

/dev/tty
/dev/tty*
/dev/console

NOTES

Plexus defines two extra flags in the *c_iflag* field of the **<termio.h>** structure. They are

```
      15     14
   ------------------------------------------------------------------------
   |     |     |     |     |              ...                          |
   |__|_|__|_|____|____|_________________________________________|
        |     |
        |     |____   ICTS
        |
        |_________   ICTSLO
```

where

ICTS specifies that Clear to Send (CTS) is to be used as flow control. Note that XON and XOFF processing is not disabled. Also, the ICP must be configured to accept the CTS signal.

ICTSLO inverts the action of the CTS activity. This is for use with devices in which the polarity of the CTS signal is inverted.

These two flags are not supported in stock SYSTEM III. The location of these flags may also change in future releases. This feature is available on ICP ports only.

The ICPs that correspond to *tty* device addresses are as follows:

/dev/tty0 - **/dev/tty7** ICP 0
/dev/tty8 - **/dev/tty15** ICP 1
/dev/tty16 - **/dev/tty23** ICP 2
/dev/tty24 - **/dev/tty31** ICP 3
/dev/tty32 - **/dev/tty39** ICP 4

At least one ICP must be reserved if you use the Virtual Protocol Machine (VPM). This must be the first ICP, or, if you use more than one VPM ICP, the first *n* ICPs.

The VPM ICP may function as a TTY ICP, although not at the same time it is being used for VPM.

If you use your VPM ICP as a TTY ICP, you will probably want two sets of **mknods**, and two each of the **/etc/inittab** and **/etc/rc** files. The idea is to switch back and forth between the sets as you move from VPM applications to TTY applications. Two sets are required because you will want to disable logins on the TTY ports of the VPM ICP when it is in use for VPM; otherwise, continuous **gettys** are done for those ports. Switching back and forth is not a trivial procedure; you must reboot each time you change over. A shell procedure incorporating all the steps makes it somewhat easier.

SEE ALSO

stty(1), ioctl(2), icp(4).

NAME

vpm - The Virtual Protocol Machine

DESCRIPTION

This entry describes a particular kind of special file and gives an introduction to the Virtual Protocol Machine (VPM).

The VPM is a software construct for implementing link protocols on the ICP in a high-level language. This is accomplished by a compiler that runs on UNIX and that translates a high-level language description of a protocol into an intermediate language that is interpreted by an interpreter running in the ICP.

The VPM driver is functionally split into two parts: a top VPM device and a bottom VPM device. The top device may be modified or replaced to suit particular applications; the bottom device interfaces with the VPM interpreter using the ICP driver. When using the *mknod* command to make a directory entry and corresponding i-node for a VPM special file, the minor device number identifies the physical ICP device, the VPM protocol number, and the physical ICP line number to be used for this special file. The two most significant bits of the minor device number denote the physical ICP device; the next two bits denote the VPM protocol number; the four least significant bits denote the VPM ICP line number. For example, if ICP device 1 is to be used with protocol number 2, which in turn is to be used with ICP device 3, the minor device number would be 0143 (octal).

UNIX user processes transfer data to or from a remote terminal or computer system through VPM using normal *open*, *read*, *write*, and *close* operations. Flow control and error recovery are provided by the protocol description residing in the ICP.

The VPM software consists of six components:

1. *vpmc*(1C): compiler for the protocol description language; it runs on UNIX.
2. VPM interpreter: a ICP program that controls the overall operation of the ICP and interprets the protocol script.
3. *si.c*: a UNIX driver that provides the interface to the VPM.
4. *vpmstart*(1C): a UNIX command that copies a load module into the ICP and starts it.
5. *vpmsnap*(1C): a UNIX command that prints a time-stamped event trace while the protocol is running.
6. *vpmtrace*(1C): a UNIX command that prints an event trace for debugging purposes while the protocol is running.

The VPM *open* for reading-and-writing is exclusive; *opens* for reading-only or writing-only are not. The VPM *open* checks that the correct interpreter is running in the ICP, then sends a RUN command to the interpreter (causing it to start interpreting the protocol script), and supplies a 512-byte receive buffer to the interpreter.

The VPM *read* returns either the number of bytes requested or the number remaining in the current receive buffer, whichever is less. Bytes remaining in a receive buffer are used to satisfy subsequent reads. The VPM *write* copies the user data into 512-byte system buffers and passes them to the VPM interpreter in the ICP for transmission.

The VPM *close* arranges for the return of system buffers and for a general cleanup when the last transmit buffer has been returned by the interpreter.

The user command *vpmtrace*(1C) reads the trace driver and prints event records. While this command is executing, the VPM driver will generate a number of event records, allowing the activity of the VPM driver and protocol script to be monitored for debugging purposes. The system functions *vpmopen*, *vpmread*, *vpmwrite*, and *vpmclose* generate event records (identified respectively by **o**, **r**, **w**, and **c**). Calls to the *vpmc*(1C) primitive *trace(arg1,arg2)* cause the VPM interpreter to pass *arg1* and *arg2* along with the current value of the script location counter to the VPM driver, which generates an event record identified by a **T**. Each event record is structured as follows:

```
struct event {
        short   e_seqn;         /*sequence number*/
        char    e_type;         /*record identifier*/
        char    e_dev;          /*minor device number*/
        short   e_short1;       /*data*/
        short   e_short2;       /*data*/
}
```

When the script terminates for any reason, the driver is notified and generates an event record identified by an **E**. This record also contains the minor device number, the script location counter, and a termination code defined as follows:

0	Normal termination; the interpreter received a *halt* command from the driver.
1	Undefined virtual-machine operation code.
2	Script program counter out of bounds.
3	Interpreter stack overflow or underflow.
4	Jump address not even.
5	MULTIBUS error.
6	Transmit buffer has an odd address; the driver tried to give the interpreter too many transmit buffers; or a *get* or *rtnxbuf* was executed while no transmit buffer was open, i.e., no *getxbuf* was executed prior to the *get* or *rtnxbuf*.
7	Receive buffer has an odd address; the driver tried to give the interpreter too many receive buffers; or a *put* or *rtnrbuf* was executed while no receive buffer was open, i.e., no *getrbuf* was executed prior to the *get* or *rtnxbuf*.
8	The script executed an *exit*.
9	A *crc16* was executed without a preceding *crcloc* execution.
10	Interpreter detected loss of modem-ready signal.
11	Transmit-buffer sequence-number error.
12	Command error; an invalid command or an improper sequence of commands was received from the driver.
13	Not used.
14	Invalid transmit state.
15	Invalid receive state.
16	Not used.
17	*Xmtctl* or *setctl* attempted while transmitter was still busy.
18	Not used.
19	Same as error code 6.
20	Same as error code 7.
21	Script to large.
22	Used for debugging the interpreter.
23	The driver's OK-check has timed out.

SEE ALSO

vpmc(1C), vpmstart(1C), trace(4).

NAME

intro - introduction to file formats

DESCRIPTION

This section outlines the formats of various files. The C **struct** declarations for the file formats are given where applicable. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

NOTES

Plexus adds *D-hosts*, for use with the Plexus Network Operating System (NOS). Plexus also adds *holidays*, *termcap*, and *ttytype* and does not currently support *master*.

NAME

a.out - assembler and link editor output

DESCRIPTION

A.out is the output file of the assembler *as* and the link editor *ld*. Both programs will make **a.out** executable if there were no errors in assembling or linking, and no unresolved external references.

This file has four sections: a header, the program text and data segments, relocation information, and a symbol table (in that order). The last two sections may be missing if the program was linked with the **-s** option of *ld*(1) or if the symbol table and relocation bits were removed by *strip*(1). Also note that if there were no unresolved external references after linking, the relocation information will be removed.

The sizes of each segment (contained in the header, discussed below) are in bytes and are even. The size of the header is not included in any of the other sizes.

When an **a.out** file is loaded into memory for execution, three logical segments are set up: the text segment, the data segment (initialized data followed by uninitialized, the latter actually being initialized to all 0's), and a stack. The text segment begins at location 0 in the core image; the header is not loaded. If the magic number (the first field in the header) is 107 (hexadecimal), it indicates that the text segment is not to be write-protected or shared, so the data segment will be contiguous with the text segment. If the magic number is 108 (hexadecimal), the data segment begins at the first 0 mod 2K byte boundary (Z8000) or the first 0 mod 4K byte boundary (MC68000) following the text segment, and the text segment is not writable by the program; if other processes are executing the same **a.out** file, they will share a single text segment. For the Z8000 only, if the magic number is 109 (hexadecimal), the text segment is again pure (write-protected and shared); moreover, the instruction and data spaces are separated. The text and data segment both begin at location 0. See the *Zilog Z8000 Instruction Manual* for restrictions that apply to this situation.

The stack will occupy the highest possible locations in the core image: on the Z8000, from FFFE (hexadecimal) and growing downwards; on the MC68000, from 1FFFFC and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *brk*(2) system call.

The start of the text segment in the **a.out** file is *hsize*; the start of the data segment is $hsize + S_t$ (the size of the text), where *hsize* is 10 (hexadecimal).

The value of a word in the text or data portions that is not a reference to an undefined external symbol is exactly the value that will appear in memory when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information (discussed below) for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added to the word in the file.

Header-Z8000

The format of the **a.out** header is as follows:

```
struct  exec      {
        short     a_magic; /* magic number */
        unsigned  a_text;  /* size of text segment */
        unsigned  a_data;  /* size of data segment */
        unsigned  a_bss;   /* size of bss segment */
        unsigned  a_syms;  /* size of symbol table */
        unsigned  a_entry; /* entry point of program */
        unsigned  a_stamp; /* version stamp */
        unsigned  a_flag;  /* set if relocation info stripped */
};
```

Header-MC68000

The format of the header on the MC68000 is as follows:

```
struct  bhdr      {
        long      fmagic;  /* magic number */
        long      tsize;   /* size of text segment */
        long      dsize;   /* size of data segment */
        long      bsize;   /* size of bss segment */
        long      ssize;   /* size of symbol table */
        long      rtsize;  /* size of text relocation info */
        long      rdsize;  /* size of data relocation info */
        long      entry;   /* entry point of program */
};
```

Relocation-Z8000

If relocation information is present, it amounts to two bytes per relocatable datum. There is no relocation information if the "suppress relocation" flag (*a_flag*) in the header is on.

The format of the relocation data is:

```
struct  r_info  {
        int     r_symbolnum:11;
                r_segment:3;
                r_pcrel:1;
};
```

The *r_pcrel* field is not used.

The *r_segment* field indicates the segment referred to by the text or data word associated with the relocation word:

- 00 indicates the reference is absolute;
- 02 indicates the reference is to the text segment;
- 04 indicates the reference is to initialized data;
- 06 indicates the reference is to bss (uninitialized data);
- 10 indicates the reference is to an undefined external symbol.

The field *r_symbolnum* contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

The start of the relocation information is

$$hsize + a\_text + a\_data$$

Relocation-MC68000

Relocation information, if it is present, is given for each datum to be relocated.

The format of the relocation information is:

```
struct  reloc   {
        unsigned rsegment:2;/* RTEXT, RDATA, RBSS, or REXTERN */
        unsigned rsize:2;         /* RBYTE, RWORD, or RLONG */
        unsigned rdisp:1;         /* 1 => a displacement */
        unsigned relpad1:3;/* unused portion of relocation tag */
        char relpad2;             /* unused portion of relocation tag */
        short rsymbol;            /* id of the symbol of external relocations */
        long rpos;                /* position of relocation in segment */
};
```

The *rsegment* field indicates the segment referred to by the relocated datum.

00 indicates the reference is to the text segment;
01 indicates the reference is to initialized data;
02 indicates the reference is to bss (uninitialized data);
03 indicates the reference is to an undefined external symbol.

The *rsize* field indicates the size of the datum:

00 indicates the datum is one byte;

01 indicates the datum is one word;

02 indicates the datum is a long.

The field *rsymbol* contains a symbol number in the case of external references. The first symbol is numbered 0, the second 1, etc. The start of the text relocation information is

tsize + *dsize* + *ssize*

The start of the data relocation information is

hsize + *tsize* + *dsize* + *ssize* + *rtsize*

Symbol Table-Z8000

The symbol table on the Z8000 consists of entries of the form:

```
struct  nlist     {
        char      n_name[8];
        int       n_type;
        unsigned  n_value;
};
```

The *n_name* field contains the ASCII name of the symbol, null-padded. The *n_type* field indicates the type of the symbol; the following values are possible:

000 undefined symbol
001 absolute symbol
002 text segment symbol
003 data segment symbol
004 bss segment symbol
037 file name symbol (produced by *ld*)
040 undefined external symbol
041 absolute external symbol
042 text segment external symbol
043 data segment external symbol
044 bss segment external symbol

The start of the symbol table on the Z8000 is:

hsize+2(*a_text*+*a_data*)

if relocation information is present, and

hsize+*a_text*+*a_data*

if it is not.

If a symbol's type is *undefined external* and the value field is non-zero, the symbol is interpreted by the link editor *ld*(1) as the name of a common region whose size is indicated by the value of the symbol.

Symbol Table-MC68000

The symbol table on the MC68000 consists of entries of the form:

```
struct  sym     {
        char    stype;          /* symbol type */
        char    sympad;         /* pad to long align */
        long    svalue;         /* value */
};
```

The symbol follows each entry and is null-terminated. The *stype* field indicates the type of the symbol; the following values are possible:

000 undefined symbol
001 absolute symbol
002 text segment symbol
003 data segment symbol
004 bss segment symbol
037 file name symbol (produced by *ld*)
024 register name
040 external bit or'd in
"%08x" format for printing a value

The start of the symbol table on the MC68000 is

hsize + *tsize* + *dsize*

If a symbol's type is *undefined external* and the value field is non-zero, the symbol is interpreted by the link editor *ld*(1) as the name of a common region whose size is indicated by the value of the symbol.

SEE ALSO

as(1), ld(1), nm(1), strip(1).

NAME

acct - per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by **<sys/acct.h>**, whose contents are:

```
/*
 * Accounting structures
 */

typedef ushort comp_t;              /* "floating point" */
                                    /* 13-bit fraction, 3-bit exponent */

struct acct
{
        char    ac_flag;            /* accounting flag */
        char    ac_stat;            /* exit status */
        ushort  ac_uid;             /* accounting user ID */
        ushort  ac_gid;             /* accounting group ID */
        dev_t   ac_tty;             /* control typewriter */
        time_t  ac_btime;           /* beginning time */
        comp_t  ac_utime;           /* acctng user time in clock ticks */
        comp_t  ac_stime;           /* acctng system time in clock ticks */
        comp_t  ac_etime;           /* acctng elapsed time in clock ticks */
        comp_t  ac_mem;             /* memory usage */
        comp_t  ac_io;              /* chars transferred */
        comp_t  ac_rw;              /* blocks read or written */
        char    ac_comm[8];         /* command name */
};

extern  struct  acct                acctbuf;
extern  struct  inode               *acctp;/* inode of accounting file */

#define AFORK   101                 /* has executed fork, but no exec */
#define ASU     02                  /* used super-user privileges */
#define ACCTF   0300                /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem*/*ac_stime* can be viewed as an approximation to the mean process size, as modified by text-sharing.

The following structure represents the total accounting format used by the various accounting commands:

```
/*
 *      total accounting (for acct period), also for day
 */

struct tacct     {
        uid_t           ta_uid;       /* userid */
        char            ta_name[8];   /* login name */
        float           ta_cpu[2];    /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2];  /* cum. kcore-minutes, p/np */
        float           ta_con[2];    /* cum. conn. time, p/np, mins */
        float           ta_du;        /* cum. disk usage */
        long            ta_pc;        /* count of processes */
        unsigned short  ta_sc;        /* count of login sessions */
        unsigned short  ta_dc;        /* count of disk samples */
        unsigned short  ta_fee;       /* fee for special services */
};
```

SEE ALSO

acct(1M), acctcom(1), acct(2).

BUGS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

ar - archive file format

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(1).

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 0177545(octal) (it was chosen to be unlikely to occur anywhere else). The header of each file is 26 bytes long:

```
#ifdef z8000
#define ARMAG       0177545
struct    ar_hdr {
          char  ar_name[14];
          long  ar_date;
          char  ar_uid;
          char  ar_gid;
          int   ar_mode;
          long  ar_size;
};
#else
#define ARMAG       "!<arch>0
#define SARMAG      8

#define ARFMAG      "'0

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];
};
#endif
```

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

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

SEE ALSO

ar(1), arcv(1), ld(1).

BUGS

The archive header structure is not compatible between the Z8000 and the 68000 due to the different word sizes. See *arcv*(1) to convert between processors.

NAME

checklist - list of file systems processed by fsck

DESCRIPTION

Checklist resides in directory **/etc** and contains a list of at most 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck*(1M) command.

SEE ALSO

fsck(1M).

NAME

core - format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See *signal*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the process's working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The important stuff not detailed therein is the locations of the registers, which are outlined in **/usr/include/sys/reg.h**.

SEE ALSO

adb(1), crash(1M), setuid(2), signal(2).

NAME

cpio - format of cpio archive

DESCRIPTION

The *header* structure, when the **c** option is not used, is:

```
struct {
        short   h_magic,
                h_dev,
                h_ino,
                h_mode,
                h_uid,
                h_gid,
                h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **c** option is used, the *header* information is described by the statement below:

```
        sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%6o%s",
                &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
                &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
                &Longtime,&Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file is recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

cpio(1), find(1), stat(2).

NAME

D-hosts - configuration file for the Network Operating System (NOS)

DESCRIPTION

The file **/usr/lib/nos/D-hosts** establishes the configuration of the Network Operating System (NOS). This file identifies the remote hosts accessible to the local host. It also specifies the protocol and physical link to be used when communicating with a given remote host.

/usr/lib/nos/D-hosts is read only at initialization time when the system is booted. Hence to locally reconfigure the network, the system must be rebooted. An error message is generated if the file cannot be located.

The file **/usr/lib/nos/D-hosts** contains one line for each remote host. This line describes various properties of the remote host. Each line is composed of a number of fields:

name:NA=xxxx:PL=ether:LL=pdlc:NL=pdlc:TL=ncf

where

name is the host name, remote or local. The name is limited to 9 characters; names longer than this are truncated. Only the characters 0-9, a-z, and A-Z may be used. The local host's name must match the 'Sys3 nodename:' as established via **dconfig**(8).

NA network address. The associated value is a hexadecimal number designating the address of the host. The number is delivered with the Ethernet controller hardware. NA values are used by the hardware drivers to route communication packets at the physical level.

PL physical level. This is the 'type' of the physical link. The associated value is a NOS-defined character string. The only physical media currently supported is *ether*.

LL link level. This is the link layer of the protocol. The associated value is a NOS-defined string. The only link level protocol currently supported is *pdlc*.

NL network level. The network layer of the protocol. The associated value is a NOS-defined character string. The only net level protocol currently supported is *pdlc*.

TL transport level. The transport layer of the protocol. The associated value is a NOS-defined character string. The only transaction level protocol currently supported is *ncf*.

A line in **/usr/lib/nos/D-hosts** may be commented by beginning it with a '#'. A line may be extended by using a '\' as the last character. This causes the EOL to be ignored, and the line may be continued on the following line. Spaces and tabs are ignored except as string and number delimiters.

DIAGNOSTICS

The following error messages may occur during boot because of an invalid configuration file:

Can not open /usr/lib/nos/D-hosts
: You have not provided a configuration file in **/usr/lib/nos**. Check to ensure that file exists. This may also be a symptom of a damaged file system.

no : <char>
: The delimiter of the fields within a descriptor is a ':'; an unknown character <char> was encountered instead of the expected ':'. Check file for bad entry or invisible characters.

no = <char>
: The assignment operator within each field is a '='; an unknown character <char> was encountered instead of the expected '='. Check file for bad entry or invisible characters.

unknown type of ncf initalization argument <string>
The configuration paramter argument was illegal. Only *NA*, *PL*, *LL*, *NL*, and *TL* are allowed.

physical layer ..." not yet implemented"
Currently only 'ether' is valid as a PL value.

Unknown host id
driver address (...) not found in configuration table
The address of the hardware was not found in the configuration table. Add an entry for your device into the file.

configuration table device name (..) does not match host nodename (..)
The host name is obtained from the disk at boot time. It does not agree with the name given the host in the configuration file. Change the configuration file or use **dconfig**(8) to change host name so that both are consistent.

NAME

dir - format of directories

SYNOPSIS

#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 *fs*(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; there is no parent, so .. has the same meaning as ..

SEE ALSO

fs(5).

NAME

dump - incremental dump tape format

DESCRIPTION

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of a header record, some bit mask records, a group of records describing file system directories, a group of records describing file system files, and some records describing a second bit mask.

The header record and the first record of each description have the format described by the structure included by

#include <dumprestor.h>

This include file has the following contents:

```
#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;
};
```

NTREC is the number of 1024 byte blocks in a physical tape record. *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_TYPE	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 mask follows. This bit mask has a one bit for each inode that was dumped.
TS_ADDR	A subblock to a file (*TS_INODE*). See the description of *c_count* below.
TS_END	End of tape record.
TS_CLRI	A bit mask follows. This bit mask contains a one bit for all inodes that were empty on the file system when dumped.
MAGIC	All header blocks have this number in *c_magic*.
CHECKSUM	Header blocks 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 block number of this record. This is counting 1024 byte blocks.
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 block sum to *CHECKSUM*.
c_dinode	This is a copy of the inode as it appears on the file system.
c_count	This is the count of characters following that describe the 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 and it is replaced as a hole in the file. If there is not sufficient space in this block to describe all of the blocks in a file, *TS_ADDR* blocks will be scattered through the file, each one picking up where the last left off.
c_addr	This is the array of characters that is used as described above.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a *TS_END* block and then the tapemark.

The structure **idates** describes an entry of the file where dump history is kept.

SEE ALSO

dump(1M), restor(1M), fs(5).

NAME

errfile - error-log file format

DESCRIPTION

When hardware errors are detected by the system, an error record is generated and passed to the error-logging daemon for recording in the error log for later analysis. The default error log is **/usr/adm/errfile**.

The format of an error record depends on the type of error that was encountered. Every record, however, has a header with the following format:

```
struct errhdr {
        int             e_type;         /* record type */
        int             e_len;          /* bytes in record (inc hdr) */
        time_t          e_time;         /* time of day */
};
```

The permissible record types are as follows:

```
#define E_GOTS   010    /* Start for UNIX 3.0*/
#define E_GORT   011    /* Start for UNIX/RT */
#define E_STOP   012    /* Stop */
#define E_TCHG   013    /* Time change */
#define E_CCHG   014    /* Configuration change */
#define E_BLK    020    /* Block device error */
#define E_STRAY  030    /* Stray interrupt */
#define E_PRTY   031    /* Memory parity */
```

Some records in the error file are of an administrative nature. These include the startup record that is entered into the file when logging is activated, the stop record that is written if the daemon is terminated "gracefully", and the time-change record that is used to account for changes in the system's time-of-day. These records have the following formats:

```
struct estart {
        struct errhdr   e_hdr;          /* record header */
        int             e_cpu;          /* CPU type */
        int             e_mmr3;         /* contents mem mgmt reg 3 */
        long            e_syssize;      /* Z8000 system memory size */
        int             e_bconf;        /* block dev configuration */
};

struct eend {
        struct errhdr   e_hdr;          /* record header */
};

struct etimchg {
        struct errhdr   e_hdr;          /* record header */
        time_t          e_ntime;        /* new time */
};
```

Stray interrupts cause a record with the following format to be logged in the file:

```
struct estray {
      struct errhdr  e_hdr;         /* record header */
      physadr        e_saddr;       /* stray loc or device addr */
      int            e_sbacty;      /* active block devices */
};
```

Memory subsystem error on 11/70 processors cause the following record to be generated:

```
struct eparity {
      struct errhdr  e_hdr;         /* record header */
      int            e_parreg[5];   /* memory subsys registers */
};
```

Error records for block devices have the following format:

```
struct eblock {
      struct errhdr  e_hdr;         /* record header */
      dev_t          e_dev;         /* "true" major + minor dev no */
      physadr        e_regloc;      /* controller address */
      int            e_bacty;       /* other block I/O activity */
      struct iostat {
        long         io_ops;        /* number read/writes */
        long         io_misc;       /* number "other" operations */
        unsigned     io_unlog;      /* number unlogged errors */
      }              e_stats;
      int            e_bflags;      /* read/write, error, etc */
      int            e_cyloff;      /* logical dev start cyl */
      daddr_t        e_bnum;        /* logical block number */
      unsigned       e_bytes;       /* number bytes to transfer */
      long           e_memadd;      /* buffer memory address */
      unsigned       e_rtry;        /* number retries */
      int            e_nreg;        /* number device registers */
};
```

The following values are used in the *e_bflags* word:

```
#define E_WRITE  0      /* write operation */
#define E_READ   1      /* read operation */
#define E_NOIO   02     /* no I/O pending */
#define E_PHYS   04     /* physical I/O */
#define E_MAP    010    /* Unibus map in use */
#define E_ERROR  020    /* I/O failed */
```

The "true" major device numbers that identify the failing device are as follows:

```
#define PD0      0
#define PT0      1
#define IS0      2
#define RM0      3
```

SEE ALSO

errdemon(1M).

NAME

file system - format of system volume

SYNOPSIS

#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>

DESCRIPTION

Every file system storage volume (e.g., RP04 disk) 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 or other information.

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

```
/*
 * Structure of the super-block
 */
struct      filsys
{
    ushort  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 */
    short   s_dinfo[4];         /* device information */
    daddr_t s_tfree;            /* total free blocks*/
    ino_t   s_tinode;           /* total free inodes */
    char    s_fname[6];         /* file system name */
    char    s_fpack[6];         /* file system pack name */
};
```

S_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize*-2 blocks long. *S_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], ..., *s_free*[*s_nfree*-1], up to 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; 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 number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

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 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

S_tinode is the total free inodes available in the file system.

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 read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

S_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long, so 16 of them fit into a block. Therefore, i-node i is located in block $(i+31)/16$, and begins $64\times((i+31) \pmod{16})$ bytes from its start. I-node 1 is reserved for future use. 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. For the format of an inode and its flags, see *inode*(5).

FILES

/usr/include/sys/filsys.h
/usr/include/sys/stat.h

NOTES

Block size is 1024 bytes, so the formulas given here for calculating the whereabouts of i-nodes are slightly different from stock SYSTEM III.

SEE ALSO

fsck(1M), fsdb(1M), mkfs(1M), inode(5).

NAME

fspec - format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on UNIX with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets <: and :>. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a **-** followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a **-** followed by the name of a "canned" tab specification.

Standard tabs are specified by **t-8**, or equivalently, **t1,9,17,25**,etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t-8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

    * <:t5,10,15 s72:> *

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

Several UNIX commands correctly interpret the format specification for a file. Among them is *gath* (see *send*(1C)) which may be used to convert files to a standard format acceptable to other UNIX commands.

SEE ALSO

ed(1), reform(1), send(1C), tabs(1).

NAME

group - group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
comma-separated list of all user 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), passwd(1), crypt(3C), passwd(5).

NAME

holidays - defining holidays and prime time for accounting

DESCRIPTION

The accounting programs *acctcon1* and *acctprc1* print usage data, dividing the data between prime and nonprime time. A holiday is nonprime time. The programs get the definition for prime and nonprime time and holidays from the file **/usr/lib/acct/holidays**. If this file is missing or the data is garbled or missing, the programs will use predefined values for the times and holidays.

A sample /usr/lib/acct/holidays file:

```
holidays = 0 44 163 188 329 330 349 350 364
prime = 9:15
nonprime = 17:00
year = 1982
```

The holidays are days of the year, starting with 0. They must be separated by white space, they must all fit on one line (maximum 200 characters), and only the first 30 holidays are used.

The prime and nonprime variables define the starting and ending times, respectively, for the prime time. The times are given in hours and minutes separated by a colon as shown.

The year is the current year. If this does not agree with the year as determined by the date command, the acctcon1 and accprc1 programs will issue a mild protest.

NOTES

This command originates from Plexus; it is not part of standard SYSTEM III UNIX.

SEE ALSO

acctcon1(1) and acctprc1(1)

NAME

inittab - control information for init

DESCRIPTION

When a state is entered, *init* reads the file **/etc/inittab**. Lines in this file have the format:

state:id:flags:command

All lines in which the *state* field match *init*'s current state are recognized. If a process is active under the same two character *id* as a recognized line, it may be terminated (signal 15), killed (signal 9), or both by including the *flags* **t** and **k** in the order desired. The signal is sent to all processes in the process group associated with the *id*. The *command* field is saved for later execution. The *flag* **c** requires the *command* to be continuously reinvoked whenever the process with that *id* dies. Otherwise the *command* is invoked a maximum of one time in the current state.

Init ignores lines with the flag "o". Note that *init* kills processes only when directed to by the "k" or "t" flags.

FILES

/etc/inittab

NAME

inode - format of an inode

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>

DESCRIPTION

An i-node for a plain file or directory in a file system has the following structure defined by **<sys/ino.h>**.

```
        /* Inode structure as it appears on a disk block. */
struct dinode
{
        ushort di_mode;      /* mode and type of file */
        short  di_nlink;     /* number of links to file */
        ushort di_uid;       /* owner's user id */
        ushort 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.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(7).

FILES

/usr/include/sys/ino.h

SEE ALSO

stat(2), fs(5), types(7).

NAME

mnttab - mounted file system table

SYNOPSIS

```
struct mnttab {
        char            mt_dev[MNTPATH];
        char            mt_node[MNTPATH];
        char            mt_filsys[MNTPATH];
        short   mt_ro_flg;
        time_t  mt_time;
};
```

DESCRIPTION

Mnttab resides in directory **/etc** and contains a table of devices mounted by the *mount*(1M) and *rmount*(1M) commands.

MNTPATH is currently 50.

Each entry is 156 bytes in length; the first 50 bytes are the null-padded name of the place where the special file or remote directory is mounted; the next 50 bytes contain the node name of the remote system when *rmount* is invoked; the next 50 bytes represent the null-padded root name of the mounted special file or remote directory; the remaining 6 bytes contain the read/write permissions of the mounted special file or remote directory, and the date on which it was mounted.

The maximum number of entries in *mnttab* is based on the system parameter **NMOUNT** located in **/usr/src/uts/cf/conf.c**, which defines the number of allowable mounted special files.

SEE ALSO

mount(1M), rmount(1M).

NAME

passwd - password file

DESCRIPTION

Passwd contains for each user the following information:

login name
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user ID
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

The encrypted password consists of 13 characters chosen from a 64 character alphabet (**.**, **/**, **0-9**, **A-Z**, **a-z**), except when the password is null in which case the encrypted password is also null. Password aging is effected for a particular user if his encrypted password in the password file is followed by a comma and a non-null string of characters from the above alphabet. (Such a string must be introduced in the first instance by the super-user.) The first character of the age, *M* say, denotes the maximum number of weeks for which a password is valid. A user who attempts to login after his password has expired will be forced to supply a new one. The next character, *m* say, denotes the minimum period in weeks which must expire before the password may be changed. The remaining characters define the week (counted from the beginning of 1970) when the password was last changed. (A null string is equivalent to zero.) *M* and *m* have numerical values in the range 0-63. If $m = M = 0$ (derived from the string **.** or **..**) the user will be forced to change his password the next time he logs in (and the "age" will disappear from his entry in the password file). If $m > M$ (signified, e.g., by the string **./**) only the super-user will be able to change the password.

FILES

/etc/passwd

SEE ALSO

login(1), passwd(1), a64l(3C), crypt(3C), getpwent(3C), group(5).

NAME

plot - graphics interface

DESCRIPTION

Files of this format are produced by routines described in *plot*(3X) and are interpreted for various devices by commands described in *tplot*(1G). 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*(3X).

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 *tplot*(1G).

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 new-line.

e erase: Start another frame of output.

f linemod: Take the following string, up to a new-line, as the style for drawing further lines. The styles are "dotted", "solid", "longdashed", "shortdashed", and "dotdashed". Effective only for the **-T4014** and **-Tver** options of *tplot*(1G) (Tektronix 4014 terminal and Versatec plotter).

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 *tplot*(1G). 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 is not square.

DASI 300	space(0, 4096, 0, 4096);
DASI 300s	space(0, 4096, 0, 4096);
DASI 450	space(0, 4096, 0, 4096);
Tektronix 4014	space(0, 3120, 0, 3120);
Versatec plotter	space(0, 2048, 0, 2048);

SEE ALSO

graph(1G), tplot(1G), plot(3X), gps(5), term(7).

NAME

pnch - file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

NAME

profile - setting up an environment at login time

DESCRIPTION

If your login directory contains a file named **.profile**, that file will be executed (via the shell's **exec .profile**) before your session begins; **.profiles** are handy for setting exported environment variables and terminal modes. If the file **/etc/profile** exists, it will be executed for every user before the **.profile**. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM LOGNAME
# Set file creation mask
umask 22
# Tell me when new mail comes in
MAIL=/usr/mail/myname
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
echo "terminal: \c"
read TERM
case $TERM in
        300)        stty cr2 nl0 tabs; tabs;;
        300s)       stty cr2 nl0 tabs; tabs;;
        450)        stty cr2 nl0 tabs; tabs;;
        hp)         stty cr0 nl0 tabs; tabs;;
        745|735)    stty cr1 nl1 -tabs; TERM=745;;
        43)         stty cr1 nl0 -tabs;;
        4014|tek)   stty cr0 nl0 -tabs ff1; TERM=4014; echo "\33;";;
        *)          echo "$TERM unknown";;
esac
```

FILES

$HOME/.profile
/etc/profile

SEE ALSO

env(1), login(1), mail(1), sh(1), stty(1), su(1), environ(7), term(7).

NAME

sccsfile - format of SCCS file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form **DDDDD** represent a five digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

@s DDDDD/DDDDD/DDDDD
@d <type> <**SCCS ID**> yr/mo/da hr:mi:se <pgmr> **DDDDD DDDDD**
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <**MR** number>
.
.
.
@c <comments> ...
.
.
.
@e

The first line (**@s**) contains the number of lines inserted/deleted/unchanged respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one **MR** number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(1) for more information on their use). Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t	<type of program>
@f v	<program name>
@f i	
@f b	
@f m	<module name>
@f f	<floor>
@f c	<ceiling>
@f d	<default-sid>
@f n	
@f j	
@f l	<lock-releases>
@f q	<user defined>

The **t** flag defines the replacement for the identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **-b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **sccsfile.5** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (*get*(1) with the **-e** keyletter). The **q** flag defines the replacement for the identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines **@t** and **@T**. The comments section typically will contain a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

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

NAME

termcap - terminal capability data base

SYNOPSIS

/usr/plx/termcap

DESCRIPTION

Termcap is a database describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3). *Termcap* describes terminals by listing a set of their capabilities, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap.*

Entries in *termcap* consist of a number of fields, separated by ':'. The first entry for each terminal gives the names that 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 is the most common abbreviation for the terminal, and the last name should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may contain blanks for readability.

CAPABILITIES

(P) padding may be specified
(P*) padding may be based on the number of 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 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: (1) Boolean capabilities, which indicate that the terminal has some particular feature; (2) numeric capabilities giving the size of the terminal or the size of particular delays; and (3) string capabilities, which give a sequence that 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, equals '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 a '*', 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 that 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 to imitate the description of a similar terminal in *termcap* and then 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 that 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. All terminals we have seen have an insert mode falling into one of these two classes.

The editor can handle both terminals that have an insert mode, and terminals that 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 erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl, kr, ku, kd,** and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0, k1, ..., k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0, l1, ..., l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the *termcap* 2 letter codes can be given in the **ko** capability, for example, :ko=cl,ll,sf,sb:, which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl, kr, ku, kd,** and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and **H** for **kh**. For example, the mime would be **:ma=^Kj^Zk^Xl:** indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than **^I** to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow '~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x**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 canceled 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), termlib(3), tset(1), vi(1), ul(1), more(1).

NOTES

The Plexus version of *termcap* is based on the one developed at the University of California at Berkeley.

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 - magnetic tape format

DESCRIPTION

The command *tp*(1) dumps files to and extracts files from magtape.

Block zero contains a copy of a stand-alone bootstrap program; see *tapeboot*(8).

Blocks 1 through 62 contain a directory of the tape. There are 496 entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct  tpent   {
        char    pathnam[32];
        short   mode;
        char    uid;
        char    uid;
        char    gid;
        char    spare;
        char    size0;
        short   size2;
        long    time;
        short   tapea;          /* tape address */
        short   unused[8];
        short   cksum;          /* check sum */
}
```

The *pathnam* entry is the path name of the file when put on the tape. If the path name 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*, the sizes and time modified are the same as described under i-nodes (*fs*(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+1023)/1024 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks 63 on are available for file storage.

A fake entry has a size of zero. See *tp*(1).

SEE ALSO

cpio(1), tp(1), fs(5), tapeboot(8).

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. The terminal kinds are from the names listed in *termcap*(5). Each port description occupies one line. The line contains the terminal kind, a space, and the name of the TTY, minus the **/dev** prefix. A sample *ttytype* file looks like this:

```
vt100 console
adm3a tty0
vt100 tty1
vt52 tty2
vt100 tty3
vt100 tty4
dm1520 tty5
vt100 tty6
vt100 tty7
```

This information is used by *tset*(1) and *login*(1) to initialize the TERM variable at login time.

SEE ALSO

tset(1), login(1).

NAME

utmp, wtmp - utmp and wtmp entry format

DESCRIPTION

The files **utmp** and **wtmp** hold user and accounting information for use by commands such as *who*(1), *acctcon1* (see *acctcon*(1M)), and *login*(1). They have the following structure, as defined by **<utmp.h>**:

```
struct utmp
{
        char    ut_line[8];         /* tty name */
        char    ut_name[8];         /* login name */
        long    ut_time;            /* time on */
};
```

FILES

/etc/utmp
/usr/adm/wtmp
/usr/include/utmp.h

SEE ALSO

acctcon(1M), login(1), who(1), write(1).

NAME

vtconf - configuration file for the NOS Virtual Terminal facility

DESCRIPTION

The file **/usr/lib/nos/vtconf** configures the Virtual Terminal Facility of the Plexus Network Operating System (NOS).

The major device number of a virtual TTY is 22. Plexus by default creates eight virtual terminal devices: four **/dev/vltty** devices, which are used by remote systems for logging in to the local system; and four **/dev/vtty** devices, which are used by the local system to connect to remote systems. There is no set limit to the number of virtual TTYs you can create. You can call them whatever you like.

NOS uses the file **/usr/lib/nos/vtconf** to determine what virtual connections are permitted on what lines. The file is read only at initialization time when the system is booted. Hence to locally reconfigure virtual TTYs, the system must be rebooted. An error message is generated if the file cannot be located.

Comments may be placed in **vtconf** by preceding them with a pound sign (#). New lines may be escaped with a backslash ("\").

For a *vltty*, the entries in **vtconf** have the form

<vtid>[-<vtid>]:<local nodename>[:<remote nodename>,<remote nodename>,...]

where "vtid" is the minor device number of the virtual TTY device, "local nodename" is the nodename of the system where this *vltty* is located, and "remote nodename" is the nodename of systems that are permitted to use this virtual TTY for logging in. Nodenames are limited to 9 characters in length. No entry in the third field or the keyword "all" mean that any remote system may use this virtual TTY. For example, the line

0-5:local

means that any remote system may use *vlttys* 0-5. The line

10:local:remote2

means that *vltty2* is dedicated for use by the remote system "remote2".

For a *vtty*, the entries in **vtconf** have the form

<vtid>[-<vtid>]:<remote nodename>

where "vtid" is the minor device number of the virtual TTY device, and "remote nodename" is the nodename of the system to which this virtual terminal is logically connected. For example, if virtual terminal vtty0 has the minor device number 0 and is logically connected to the system "remote1", the **vtconf** entry for vtty0 would read

0:remote1

This means that vtty0 is connected to the system "remote1". Each *vtty* may be logically connected to one and only one remote system.

EXAMPLE

If **/dev** contains these lines

```
crw-rw-rw- 1 root    22,  5 Dec 11 15:14 glenvtty
crw-rw-rw- 1 root    22,  3 Dec 11 15:14 gregvtty
crw-rw-rw- 1 root    22,  6 Jan 28 15:57 guestvtty
crw-rw-rw- 1 root    22,  4 Dec 11 15:14 jsevttty
crw-rw-rw- 1 root    22,  1 Feb 11 14:06 montevtty
crw-rw-rw- 1 root    22,  2 Feb 14 14:13 pafvtty
crw-rw-rw- 1 root    22,  0 Jan 20 16:28 sandyvtty
crw--w--w- 1 root    22, 12 Feb 15 10:37 vltty0
crw--w--w- 1 root    22, 13 Feb 14 17:58 vltty1
crw--w--w- 1 root    22, 14 Jan 11 15:27 vltty2
crw--w--w- 1 root    22, 15 Jan 10 18:26 vltty3
crw--w--w- 1 root    22, 16 Dec 18 21:08 vltty4
```

there are seven *vtty* devices (minor device numbers 0-6), and five *vltty* devices (minor device numbers 12-16); and the **vtconf** file might look like this:

```
0:remote1
1:remote2
2-6:remote3
12:local:remote1
13-14:local:remote2
15-16:local
```

The first three lines apply to *vtty* devices. The first line means that the device *sandyvtty* (minor device number 0) must be used to connect with the remote system "remote1". The second line means the device *montevtty* (minor device number 1) must be used to connect with the remote system "remote2". The third line means any of the devices *pafvtty*, *gregvtty*, *jsevtty*, *glenvtty*, or *guestvtty* may be used to connect with the remote system "remote3".

The rest of the lines apply to *vltty* devices. The fourth line means that the device *vltty0* may receive logins from remote system "remote1" only; the fifth line says that devices *vltty1* and *vltty2* will receive logins from the remote system "remote2" only. The sixth line says that the rest of the *vltty* devices will receive logins from any remote system.

DIAGNOSTICS

bad or duplicate line in vtconf line count = <line where error occurred>
Last part is <characters in line to the right of the error>
The input line contains a parse error. A parse error occurs when there is a duplicate line, or the same *vtty* is connected to more than one remote system, or the same virtual terminal is declared to be both a *vtty* and a *vltty*.

cannot build host list for VT: no space in siocbuf
The buffer for the storage of host names has overflowed.

Max no. vtty's: <number of new entries that have been added to parse table>
vtconf lists more which are ignored
The parser ran out of space trying to update **vtconf**. It was able to add some new entries, but found it could no longer access the old ones. This means there are too many virtual terminal devices.

NAME

intro - introduction to games

DESCRIPTION

This section describes the recreational and educational programs found in the directory **/usr/games**. A suggested procedure is to disallow their use during business hours by means of *cron*(1M).

NOTES

Plexus adds *fish*. The following games are not currently supported: *chess*, *maze*, *quiz*, *reversi*, and *sky*.

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**, and / 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

back - the game of backgammon

SYNOPSIS

/usr/games/back

DESCRIPTION

Back is a program which provides a partner for the game of backgammon. It is designed to play at three different levels of skill, one of which you must select. In addition to selecting the opponent's level, you may also indicate that you would like to roll your own dice during your turns (for the superstitious players). You will also be given the opportunity to move first. The practice of each player rolling one die for the first move is not incorporated.

The points are numbered 1-24, with 1 being white's extreme inner table, 24 being brown's inner table, 0 being the bar for removed white pieces and 25 the bar for brown. For details on how moves are expressed, type **y** when *back* asks "Instructions?" at the beginning of the game. When *back* first asks "Move?", type **?** to see a list of move options other than entering your numerical move.

When the game is finished, *back* will ask you if you want the log. If you respond with **y**, *back* will attempt to append to or create a file **back.log** in the current directory.

FILES

/usr/games/lib/backrules	rules file
/tmp/b*	log temp file
back.log	log file

BUGS

The only level really worth playing is "expert", and it only plays the forward game.
Back will complain loudly if you attempt to make too *many* moves in a turn, but will become very silent if you make too *few*.
Doubling is not implemented.
Back does not provide instructions.

NAME

bj - the game of black jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

Bj is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is $2 every hand.

A player "natural" (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a "push" (no money exchange).

If the dealer has an ace up, the player is allowed to make an "insurance" bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.

If the player is dealt two cards of the same value, he is allowed to "double". He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)

If a dealt hand has a total of ten or eleven, the player may "double down". He may double the bet ($2 to $4) and receive exactly one more card on that hand.

Under normal play, the player may "hit" (draw a card) as long as his total is not over twenty-one. If the player "busts" (goes over twenty-one), the dealer wins the bet.

When the player "stands" (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.

If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by **y** followed by a new-line for "yes", or just new-line for "no".

? (means, "do you want a hit?")
Insurance?
Double down?

Every time the deck is shuffled, the dealer so states and the "action" (total bet) and "standing" (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

NAME

craps - the game of craps

SYNOPSIS

/usr/games/craps

DESCRIPTION

Craps is a form of the game of craps that is played in Las Vegas. The program simulates the *roller*, while the user (the *player*) places bets. The player may choose, at any time, to bet with the roller or with the *House*. A bet of a negative amount is taken as a bet with the House, any other bet is a bet with the roller.

The player starts off with a "bankroll" of $2,000.

The program prompts with:

bet?

The bet can be all or part of the player's bankroll. Any bet over the total bankroll is rejected and the program prompts with "bet?" until a proper bet is made.

Once the bet is accepted, the roller throws the dice. The following rules apply (the player wins or loses depending on whether the bet is placed with the roller or with the House; the odds are even). The *first* roll is the roll immediately following a bet.

1. On the first roll:

7 or 11	wins for the roller;
2, 3, or 12	wins for the House;
any other number	is the *point*, roll again (Rule 2 applies).

2. On subsequent rolls:

point	roller wins;
7	House wins;
any other number	roll again.

If a player loses the entire bankroll, the House will offer to lend the player an additional $2,000. The program will prompt:

marker?

A "yes" (or "y") consummates the loan. Any other reply terminates the game.

If a player owes the House money, the House reminds the player, before a bet is placed, how many markers are outstanding.

If, at any time, the bankroll of a player who has outstanding markers exceeds $2,000, the House asks:

Repay marker?

A reply of "yes" (or "y") indicates the player's willingness to repay the loan. If only 1 marker is outstanding, it is immediately repaid. However, if more than 1 marker are outstanding, the House asks:

How many?

markers the player would like to repay. If an invalid number is entered (or just a carriage return), an appropriate message is printed and the program will prompt with "How many?" until a valid number is entered.

If a player accumulates 10 markers (a total of $20,000 borrowed from the House), the program informs the player of the situation and exits.

Should the bankroll of a player who has outstanding markers exceed $50,000, the *total* amount of money borrowed will be *automatically* repaid to the House.

Any player who accumulates $100,000 or more breaks the bank. The program then prompts:

New game?

to give the House a chance to win back its money.

Any reply other than "yes" is considered "no" (except in the case of "bet?" or "How many?"). To exit, send an interrupt (break), DEL, or control-D. The program will indicate whether the player won, lost, or broke even.

MISCELLANEOUS

The random number generator for the die numbers uses the seconds from the time of day. Depending on system usage, these numbers, at times, may seem strange but occurrences of this type in a real dice situation are not uncommon.

NAME

fish - the game of fish

SYNOPSIS

/usr/games/fish

DESCRIPTION

Fish simulates the children's card game. The player is dealt seven cards; the computer also has a hand of seven cards, which the player never sees. The players take turns asking each other if each has a certain card; e.g., "Do you have any 8's?" The player asking must have at least one of the card in question. If the other player has one or more of the cards, he must surrender them; otherwise, he draws one from the deck. The goal is to accumulate all four of each card, i.e., all the aces, all the 2's, all the 3's, and so on. Whoever has the most complete sets wins.

NAME

hangman - guess the word

SYNOPSIS

/usr/games/hangman [arg]

DESCRIPTION

Hangman chooses a word at least seven letters long from a dictionary. The user is to guess letters one at a time.

The optional argument *arg* names an alternate dictionary.

FILES

/usr/lib/w2006

BUGS

Hyphenated compounds are run together.

NAME

moo - guessing game

SYNOPSIS

/usr/games/moo

DESCRIPTION

Moo is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A "cow" is a correct digit in an incorrect position. A "bull" is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

NAME
ttt - tic-tac-toe

SYNOPSIS
/usr/games/ttt

DESCRIPTION
Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

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 Adventure.

NAME

intro - introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NOTES

Plexus continues to provide the *ms* macro package.

NAME

ascii - map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, 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

environ - user environment

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*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (:). *Login*(1) sets **PATH=:/bin:/usr/bin**.

HOME Name of the 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 *mm*(1) or *tplot*(1G), which may exploit special capabilities of that terminal.

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

LOGNAME User's login id.

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL**, **PS1**, **PS2**, **IFS**.

SEE ALSO

env(1), login(1), sh(1), exec(2), getenv(3C), profile(5), term(7).

NAME

fcntl - file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#defineO_RDONLY  0
#defineO_WRONLY  1
#defineO_RDWR    2
#defineO_NDELAY  04       /* Non-blocking I/O */
#defineO_APPEND  010      /* append (writes guaranteed at the end) */

/* Flag values accessible only to open(2) */
#defineO_CREAT   00400    /* open with file create (uses third open arg)*/
#defineO_TRUNC   01000    /* open with truncation */
#defineO_EXCL    02000    /* exclusive open */

/* fcntl(2) requests */
#defineF_DUPFD   0        /* Duplicate fildes */
#defineF_GETFD   1        /* Get fildes flags */
#defineF_SETFD   2        /* Set fildes flags */
#defineF_GETFL   3        /* Get file flags */
#defineF_SETFL   4        /* Set file flags */
```

SEE ALSO

fcntl(2), open(2).

NAME

greek - graphics for the 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 TELETYPE® Model 37 terminals equipped with a 128-character type-box. These are the default greek characters produced by *nroff*(1). 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	ς	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	∫	^			

FILES

/usr/pub/greek

SEE ALSO

300(1), 4014(1), 450(1), greek(1), hp(1), tc(1), troff(1).

NAME

man - macros for formatting entries in this manual

SYNOPSIS

nroff -man files

troff -man [**-rs1**] files

DESCRIPTION

These *troff*(1) macros are used to lay out the format of the entries of this manual. A skeleton entry may be found in the file **/usr/man/man0/skeleton**. These macros are used by the *man*(1) command.

The default page size is 8.5´´×11´´, with a 6.5´´×10´´ text area; the **-rs1** option reduces these dimensions to 6´´×9´´ and 4.75´´×8.375´´, respectively; this option (which is *not* effective in *nroff*(1)) also reduces the default type size from 10-point to 9-point, and the vertical line spacing from 12-point to 10-point. The **-rV2** option may be used to set certain parameters to values appropriate for certain Versatec printers: it sets the line length to 82 characters, the page length to 84 lines, and it inhibits underlining; this option should not be confused with the **-Tvp** option of the *man*(1) command, which is available at some UNIX sites.

Any *text* argument below may be one to six "words". Double quotes ("") may be used to include blanks in a "word". If *text* is empty, the special treatment is applied to the next line that contains text to be printed. For example, **.I** may be used to italicize a whole line, or **.SM** followed by **.B** to make small bold text. By default, hyphenation is turned off for *nroff*, but remains on for *troff*.

Type font and size are reset to default values before each paragraph and after processing font- and size-setting macros, e.g., **.I**, **.RB**, **.SM**. Tab stops are neither used nor set by any macro except **.DT** and **.TH**.

Default units for indents *in* are ens. When *in* is omitted, the previous indent is used. This remembered indent is set to its default value (7.2 ens in *troff*, 5 ens in *nroff*-this corresponds to 0.5´´ in the default page size) by **.TH**, **.PP**, and **.RS**, and restored by **.RE**.

.TH *t s c n*	Set the title and entry heading; *t* is the title, *s* is the section number, *c* is extra commentary, e.g., "local", *n* is new manual name. Invokes **.DT** (see below).
.SH *text*	Place subhead *text*, e.g., **SYNOPSIS**, here.
.SS *text*	Place sub-subhead *text*, e.g., **Options**, here.
.B *text*	Make *text* bold.
.I *text*	Make *text* italic.
.SM *text*	Make *text* 1 point smaller than default point size.
.RI *a b*	Concatenate roman *a* with italic *b*, and alternate these two fonts for up to six arguments. Similar macros alternate between any two of roman, italic, and bold: **.IR .RB .BR .IB .BI**
.P	Begin a paragraph with normal font, point size, and indent. **.PP** is a synonym for **.P**.
.HP *in*	Begin paragraph with hanging indent.
.TP *in*	Begin indented paragraph with hanging tag. The next line that contains text to be printed is taken as the tag. If the tag does not fit, it is printed on a separate line.
.IP *t in*	Same as **.TP** *in* with tag *t*; often used to get an indented paragraph without a tag.
.RS *in*	Increase relative indent (initially zero). Indent all output an extra *in* units from the current left margin.
.RE *k*	Return to the *k*th relative indent level (initially, *k*=1; *k*=0 is equivalent to *k*=1); if *k* is omitted, return to the most recent lower indent level.
.PM *m*	Produces proprietary markings; where *m* may be **P** for **PRIVATE**, **N** for **NOTICE**, **BP** for **BELL LABORATORIES PROPRIETARY**, or **BR** for **BELL LABORATORIES RESTRICTED**.

.DT Restore default tab settings (every 7.2 ens in *troff*, 5 ens in *nroff*).

.PD *v* Set the interparagraph distance to *v* vertical spaces. If *v* is omitted, set the interparagraph distance to the default value (0.4v in *troff*, 1v in *nroff*).

The following *strings* are defined:

***R** ® in *troff*(1), **(Reg.)** in *nroff*(1).

***S** Change to default type size.

The following *number registers* are given default values by **.TH**:

IN Left margin indent relative to subheads (default is 7.2 ens in *troff*, 5 ens in *nroff*).

LL Line length including **IN**.

PD Current interparagraph distance.

CAVEATS

In addition to the macros, strings, and number registers mentioned above, there are defined a number of *internal* macros, strings, and number registers. Except for names predefined by *troff*(1) and number registers **d**, **m**, and **y**, all such internal names are of the form *XA*, where *X* is one of **)**, **]**, and **}**, and *A* stands for any alphanumeric character.

If a manual entry needs to be preprocessed by *cw*(1), *eqn*(1) (or *neqn*), and/or *tbl*(1), it must begin with a special line (described in *man*(1)), causing the *man* command to invoke the appropriate preprocessor(s).

The programs that prepare the Table of Contents and the Permuted Index for this Manual assume the *NAME* section of each entry consists of a single line of input that has the following format:

name[, name, name ...] \- explanatory text

The macro package increases the inter-word spaces (to eliminate ambiguity) in the *SYNOPSIS* section of each entry.

The macro package itself uses only the roman font (so that one can replace, for example, the bold font by the constant-width font-see *cw*(1)). Of course, if the input text of an entry contains requests for other fonts (e.g., **.I**, **.RB**, **\fI**), the corresponding fonts must be mounted.

FILES

/usr/lib/tmac/tmac.an
/usr/lib/macros/cmp.[nt].[dt].an
/usr/lib/macros/ucmp.[nt].an
/usr/man/man0/skeleton

SEE ALSO

man(1), troff(1).

BUGS

If the argument to **.TH** contains *any* blanks and is *not* enclosed by double quotes (""), there will be bird-dropping-like things on the output.

NAME

mm - the MM macro package for formatting documents

SYNOPSIS

mm [options] [files]

nroff -mm [options] [files]

nroff -cm [options] [files]

mmt [options] [files]

troff -mm [options] [files]

troff -cm [options] [files]

DESCRIPTION

This package provides a formatting capability for a very wide variety of documents. It is the standard package used by the BTL typing pools and documentation centers. The manner in which a document is typed in and edited is essentially independent of whether the document is to be eventually formatted at a terminal or is to be phototypeset. See the references below for further details.

The **-mm** option causes *nroff*(1) and *troff*(1) to use the non-compacted version of the macro package, while the **-cm** option results in the use of the compacted version, thus speeding up the process of loading the macro package.

FILES

/usr/lib/tmac/tmac.m	pointer to the non-compacted version of the package
/usr/lib/macros/mm[nt]	non-compacted version of the package
/usr/lib/macros/cmp.[nt].[dt].m	compacted version of the package
/usr/lib/macros/ucmp.[nt].m	initializers for the compacted version of the package

SEE ALSO

mm(1), mmt(1), troff(1).
MM-Memorandum Macros by D. W. Smith and J. R. Mashey.
Typing Documents with MM by D. W. Smith and E. M. Piskorik.

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 and 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.
.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

mv - a macro package for making view graphs

SYNOPSIS

mvt [options] [files]
troff -mv [options] [files]

DESCRIPTION

This package provides an easy-to-use facility for making view graphs and projection slides in a variety of formats. A dozen or so macros are provided that accomplish most of the formatting tasks needed in making transparencies. All of the facilities of *troff*(1), *eqn*(1), and *tbl*(1) are available for more difficult tasks. The output can be previewed on most terminals, and, in particular, on the Tektronix 4014 and on the Versatec printer. See the reference below for further details.

FILES

/usr/lib/tmac/tmac.v

SEE ALSO

eqn(1), mvt(1), tbl(1), troff(1).
A Macro Package for View Graphs and Slides by T. A. Dolotta and D. W. Smith (in preparation).

NAME

regexp - regular expression compile and match routines

SYNOPSIS

#define INIT <declarations>
#define GETC() <getc code>
#define PEEKC() <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;

int step(string, expbuf)
char *string, *expbuf;

DESCRIPTION

This page describes general purpose regular expression matching routines in the form of *ed*(1), defined in **/usr/include/regexp.h**. Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC() — Return the value of the next character in the regular expression pattern. Successive calls to GETC() should return successive characters of the regular expression.

PEEKC() — Return the next character in the regular expression. Successive calls to PEEKC() should return the same character (which should also be the next character returned by GETC()).

UNGETC(*c*) — Cause the argument *c* to be returned by the next call to GETC() (and PEEKC()). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC(). The value of the macro UNGETC(*c*) is always ignored.

RETURN(*pointer*) — This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.

ERROR(*val*) — This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

ERROR	MEANING
11	Range endpoint too large.
16	Bad number.
25	"\digit" out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the *compile* routine is as follows:

compile(instring, expbuf, endbuf, eof)

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for programs that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more that the highest address that the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call to ERROR(50) is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a /.

Each programs that includes this file must have a **#define** statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. Most often it is used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC(), PEEKC() and UNGETC(). Otherwise it can be used to declare external variables that might be used by GETC(), PEEKC() and UNGETC(). See the example below of the declarations taken from *grep*(1).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

step(string, expbuf)

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns a one indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called, simply call *advance*.

When *advance* encounters a * or **\{ \}** sequence in the regular expression it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or **\{ \}**. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used be *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are trivial and are called by the routines previously mentioned.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr()

#include <regexp.h>
...
            compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
            if(step(linebuf, expbuf))
                        succeed();
```

FILES

/usr/include/regexp.h

SEE ALSO

ed(1), grep(1), sed(1).

BUGS

The handling of *circf* is kludgy.
The routine *ecmp* is equivalent to the Standard I/O routine *strncmp* and should be replaced by that routine.
The actual code is probably easier to understand than this manual page.

NAME

stat - data returned by stat system call

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

DESCRIPTION

The system calls *stat* and *fstat*(2) return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

```
/*
 * Structure of the result of stat
 */

struct  stat
{
        dev_t     st_dev;
        ino_t     st_ino;
        ushort    st_mode;
        short     st_nlink;
        ushort    st_uid;
        ushort    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_IFIFO   0010000 /* fifo */
#define S_ISUID   04000             /* set user id on execution */
#define S_ISGID   02000             /* set group id on execution */
#define S_ISVTX   01000             /* save swapped text even after use */
#define S_IREAD   00400             /* read permission, owner */
#define S_IWRITE  00200             /* write permission, owner */
#define S_IEXEC   00100             /* execute/search permission, owner */
```

FILES

/usr/include/sys/types.h
/usr/include/sys/stat.h

SEE ALSO

stat(2).

NAME

term - conventional names

DESCRIPTION

These names are used by certain commands (e.g., *nroff*(1), *mm*(1), *man*(1), *tabs*(1)) and are maintained as part of the shell environment (see *sh*(1), *profile*(5), and *environ*(7)) in the variable **$TERM**:

1520	Datamedia 1520
1620	Diablo 1620 and others using the HyType II printer
1620-12	same, in 12-pitch mode
2621	Hewlett-Packard HP2621 series
2631	Hewlett-Packard 2631 line printer
2631-c	Hewlett-Packard 2631 line printer - compressed mode
2631-e	Hewlett-Packard 2631 line printer - expanded mode
2640	Hewlett-Packard HP2640 series
2645	Hewlett-Packard HP264n series (other than the 2640 series)
300	DASI/DTC/GSI 300 and others using the HyType I printer
300-12	same, in 12-pitch mode
300s	DASI/DTC/GSI 300s
382	DTC 382
300s-12	same, in 12-pitch mode
3045	Datamedia 3045
33	TELETYPE® Model 33 KSR
37	TELETYPE Model 37 KSR
40-2	TELETYPE Model 40/2
4000A	Trendata 4000A
4014	Tektronix 4014
43	TELETYPE Model 43 KSR
450	DASI 450 (same as Diablo 1620)
450-12	same, in 12-pitch mode
735	Texas Instruments TI735 and TI725
745	Texas Instruments TI745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences
hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
tn1200	General Electric TermiNet 1200
tn300	General Electric TermiNet 300
vt100	Digital VT100

Up to 8 characters, chosen from [-a-z0-9], make up a basic terminal name. Terminal sub-models and operational modes are distinguished by suffixes beginning with a **-**. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept arguments of the form **-T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **$TERM**, which, in turn, should contain *term*.

SEE ALSO

mm(1), nroff(1), tplot(1G), sh(1), stty(1), tabs(1), profile(5), environ(7).

BUGS

This is a small candle trying to illuminate a large, dark problem. Programs that ought to adhere to this nomenclature do so somewhat fitfully.

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 { int r[1]; } *   physadr;
typedef long                     daddr_t;
typedef char *                   caddr_t;
typedef unsigned short           ushort;
typedef ushort                   ino_t;
#ifdef m68
typedef short                    cnt_t;
#else
typedef char                     cnt_t;
#endif
typedef long                     time_t;
#ifndef OVKRNL
#ifdef m68
typedef int                      label_t[13];    /* a2-a7, d2-d7, & pc */
#else
typedef int                      label_t[9];     /* program status regs r7 - r15 */
#endif
#else
typedef int                      label_t[10];
#endif
typedef short                    dev_t;
typedef long                     off_t;
typedef long                     paddr_t;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(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

fs(5).

NAME

intro - introduction to system maintenance procedures

DESCRIPTION

This section outlines certain procedures that will be of interest to those charged with the task of system maintenance. Included are discussions on such topics as recovery from crashes, file backups, etc.

NOTES

Plexus added the commands *autoboot*, *dconfig*, *dformat*, *fbackup*, and *gettytab*. The commands *70boot, diskboot*, *etp*, *hasp*, *romboot*, *rp6fmt*, *tapeboot*, *unixboot*, *uvac*, and *vaxops* have been eliminated. In most cases, the deleted commands apply to non-Plexus hardware.

Because of its frequent use in system maintenance, the *shutdown* command has been moved to this section from Section 1.

BUGS

No manual can take the place of good, solid experience.

NAME

autoboot - automatic reboot

DESCRIPTION

Plexus UNIX can be configured to boot itself from a system reset or power failure and go into multi-user or single-user state. If *autoboot* is enabled, the software:

1) boots the default system as set by the standalone programs **dconfig** or **dformat**, and

2) goes into state 8, the autoboot state. In state 8, the file **/etc/rc** does an **/etc/fsck** on the default disk devices and then goes into state 2, multi-user.

To enable *autoboot*, set switch S4 of the processor board switchpak to the ON position. See the section titled "Processor Board Options" in the appendix titled "Board-Level Configuration Options" in the *Plexus User's Manual*.

NOTES

This is a Plexus command. It is not part of standard SYSTEM III.

SEE ALSO

init(8), rc(8).

BUGS

While *autoboot* can ensure a running system when no one is around to reboot after a crash, it does not resolve the cause of the crash or fix any damage done by it.

NAME

crash - what to do when the system crashes

DESCRIPTION

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

How to bring it back up. If the reason for the crash is not evident (see below for guidance on "evident") you may want to try to dump the system if you feel up to debugging. At the moment a dump can be taken only on magtape. With a tape mounted and ready, stop the machine, load address 44(8) (on the PDP-11), 400(16) (on the VAX-11/780; see *vaxops*(8)), and start. This should write a copy of all of core on the tape with an EOF mark. Be sure the ring is in, the tape is ready, and the tape is clean and new.

In restarting after a crash, always bring up the system single-user, as specified in *unixboot*(8) as modified for your particular installation. Then perform an *fsck*(1M) on all file systems which could have been in use at the time of the crash. If any serious file system problems are found, they should be repaired. When you are satisfied with the health of your disks, check and set the date if necessary, then come up multi-user.

To even boot UNIX at all, three files (and the directories leading to them) must be intact. First, the initialization program **/etc/init** must be present and executable. If it is not, the CPU will loop in user mode at location 6(8) (PDP-11), 13(16) (VAX-11/780). For *init* to work correctly, **/dev/console** and **/bin/sh** must be present. If either does not exist, the symptom is best described as thrashing. *Init* will go into a *fork/exec* loop trying to create a Shell with proper standard input and output.

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

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

Fsck(1M) is adept at diagnosing and repairing file system problems. It first identifies all of the files that contain bad (out of range) blocks or blocks that appear in more than one file. Any such files are then identified by name and *fsck* requests permission to remove them from the file system. Files with bad blocks should be removed. In the case of duplicate blocks, all of the files except the most recently modified should be removed. The contents of the survivor should be checked after the file system is repaired to ensure that it contains the proper data. (Note that running *fsck* with the **-n** option will cause it to report all problems without attempting any repair.)

Fsck will also report on incorrect link counts and will request permission to adjust any that are erroneous. In addition, it will reconnect any files or directories that are allocated but have no file system references to a "lost+found" directory. Finally, if the free list is bad (out of range, missing, or duplicate blocks) *fsck* will, with the operators concurrence, construct a new one.

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

blkdev

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

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

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

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

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

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

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

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

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

PDP-11:

0	bus error
1	illegal instruction
2	BPT/trace
3	IOT
4	power fail
5	EMT
6	recursive system call (TRAP instruction)
7	11/70 cache parity, or programmed interrupt
8	floating point trap
9	segmentation violation

VAX-11/780:

0	reserved addressing fault
1	illegal instruction
2	BPT instruction trap
3	XFC instruction trap
4	reserved operand fault
5	recursive system call (CHMK instruction)
6	floating point trap
7	software level 1 (reschedule) trap
8	segmentation violation
9	protection fault
10	trace trap
11	compatibility mode fault

In some of these cases it is possible for octal 40 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred. If you wish to examine the stack after such a trap, either dump the system, or use the console switches to examine core; the required address mapping is described below.

Interpreting dumps. All file system problems should be taken care of before attempting to look at dumps. The dump should be read into the file **/usr/tmp/core**; *cp*(1) will do. At this point, you should execute *ps -el -c /usr/tmp/core* and *who* to print the process table and the users who were on at the time of the crash.

Additional information for the PDP-11. You should dump (*adb*(1)) the first 30 bytes of **/usr/tmp/core**. Starting at location 4, the registers R0, R1, R2, R3, R4, R5, SP and KDSA6 (KISA6 for 11/40s) are stored. If the dump had to be restarted, R0 will not be correct. Next, take the value of KA6 (location 22(8) in the dump) multiplied by 100(8) and dump 2000(8) bytes starting from there. This is the per-process data associated with the process running at the time of the crash. Relabel the addresses 140000 to 141776. R5 is C's frame or display pointer. Stored at (R5) is the old R5 pointing to the previous stack frame. At (R5)+2 is the saved PC of the calling procedure. Trace this calling chain until you obtain an R5 value of 141756, which is where the user's R5 is stored. If the chain is broken, you have to look for a plausible R5, PC pair and continue from there. Each PC should be looked up in the system's name list using *adb*(1) and its : command, to get a reverse calling order. In most cases this procedure will give an idea of what is wrong. A more complete discussion of system debugging is impossible.

SEE ALSO

adb(1), fsck(1M), unixboot(8), vaxops(8).

NAME

dconfig - configure logical disks

SYNOPSIS

/etc/dconfig - for use under UNIX

dconfig - for running program from release tape only

/stand/dconfig - for standalone use (UNIX not running) only

DESCRIPTION

Dconfig allows you to change the Sys3 default logical disk address assignments and the default UNIX device mapping. It also can be used to verify the logical disk configuration, change the system nodename for **uucp** and **uname**, or change the primary bootname.

Dconfig has both regular (**/etc/dconfig**) and standalone (**/stand/dconfig**) versions. Plexus release tapes also contain a copy of **dconfig**. The arguments to **/etc/dconfig** (the regular version) differ from those for the standalone and tape versions. **/etc/dconfig** expects the special files defined in the /**dev** directory as arguments, while the standalone version and the release tape version both use built-in special filenames as described in the *Plexus User's Manual*.

Dconfig prompts for responses, and gives the current values for each parameter in brackets. A <return> leaves the values the same; a <return> in response to a yes or no question defaults to "no". Unlike most Sys3 programs, **dconfig** expects response in terms of 512-byte sectors, rather than 1024 byte blocks.

Dconfig asks "Disk?". If **dconfig** for any reason (e.g., permissions) cannot access the disk you type, it continues to give the "Disk?" prompt. For complete information and examples, see the *Plexus User's Manual*.

NOTES

This is a Plexus command. It is not part of stock SYSTEM III.

SEE ALSO

uname(1).

BUGS

/etc/dconfig should be used only to examine and not change data.

NAME

dformat - disk formatter

SYNOPSIS

dformat - for running the program from a release tape only

/stand/dformat - for standalone use (no UNIX) only

DESCRIPTION

Dformat is the Sys3 disk formatting program. One option to **dformat** formats the disk and spares bad sectors; another just spares bad sectors. Other options differ for the P/40 and P/25; all are explained in detail in the *Plexus User's Manual.*

Dformat prompts for the parameters it needs. For examples, see the *Plexus User's Manual.*

NOTES

This is a Plexus command. It is not part of standard SYSTEM III.

SEE ALSO

Plexus User's Manual

NAME

fbackup - make a fast tape backup of a file system

SYNOPSIS

fbackup - for running the program from a release tape only

/stand/fbackup - for standalone (no UNIX) use only

DESCRIPTION

The standalone program **fbackup** makes a fast copy of data on disk to tape, or data on tape to disk. It is usually used to make a copy of a file system. **Fbackup** is faster than **dump** and writes in a format that is understood by **dd** (i.e., it is a byte-by-byte copy), so you should use **fbackup** rather than **dump** if you need the speed. See NOTES below for when to use which of the syntax descriptions above.

Fbackup prompts for its arguments. It can copy between an iSBC disk and 9-track tape or between an IMSC disk and cartridge. It does not support copies between an IMSC disk and 9-track tape or an iSBC disk and a cartridge. **Fbackup** writes to 9-track tape in block sizes of 16K bytes per record.

To use **fbackup**, you need to know the starting disk address of the file system, and its length in 512-byte disk sectors. To find this out, use **dconfig**(8).

On P/40 systems with the cartridge tape option, **fbackup** writes to the cartridge tape, not the 9-track tape.

NOTES

This is a Plexus program. It is not part of standard SYSTEM III.

SEE ALSO

Plexus User's Manual

BUGS

Fbackup accepts unsupported combinations of disk and tape and proceeds to copy between a supported combination.

NAME

filesave, tapesave - daily/weekly UNIX file system backup

SYNOPSIS

/etc/filesave.?
/etc/tapesave

DESCRIPTION

These shell scripts are provided as models. They are designed to provide a simple, interactive operator environment for file backup. **Filesave.?** is for daily disk-to-disk backup and **tapesave** is for weekly disk-to-tape.

The suffix **.?** can be used to name another system where two (or more) machines share disk drives (or tape drives) and one or the other of the systems is used to perform backup on both.

SEE ALSO

shutdown(8), volcopy(1M).

NAME

getty - set the modes of a terminal

SYNOPSIS

/etc/getty name type delay

DESCRIPTION

Getty is normally invoked by *init*(8) as the first step in allowing users to login to the system. Lines in **/etc/inittab** tell *init* to invoke *getty* with the proper arguments.

Name should be the name of a terminal in **/dev** (e.g., **tty03**); *type* should be a single character chosen from **-**, **0**, **1**, **2**, **3**, **4**, **5**, or **6** (may vary locally) which selects a speed table in *getty*, or **!**, which tells *getty* to update **/etc/utmp** and exit; *delay* is relevant for dial-up ports only. It specifies the time in seconds that should elapse before the port is disconnected if the user does not respond to the **login:** request.

First, *getty* types the **login:** message. The **login:** message depends on the speed table being used, and may include the characters that put the GE TermiNet 300 terminal into full-duplex, take the DASI terminals out of the plot mode, or put a TELETYPE® Model 37 into full-duplex. Then the user's login name is read, a character at a time.

While reading, *getty* tries to adapt to the terminal, speed, and mode that is being used. If a null character is received, it is assumed to be the result of a "break" ("interrupt"). The speed is then changed based on the speed table that *getty* is using, and **login:** is typed again. Subsequent breaks cause a cycling through the speeds in the speed table being used.

The user's login name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately. If the login name contains only upper-case alphabetic characters, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, *login*(1) is called with the user's login name as argument.

Speed sequences for the speed tables:

-	B110; for 110 baud console TTY.
0	B300-B150-B110-B1200; normal dial-up sequence starting at B300.
1	B150; no sequence.
2	B2400; no sequence.
3	B1200-B300-B150-B110; normal dial-up sequence starting at B1200.
4	B300; for console DECwriter.
5	B9600; no sequence.
6	B4800-B9600; for Tektronix 4014.
7	B4800; no sequence.
a	External a, 19.2K baud, no sequence.
b	External b, baud rate determined by external switches, no sequence.

Additional speed tables can be defined via *gettytab.c*. See *gettytab*(8).

SEE ALSO

login(1), tty(4), inittab(5), utmp(5), init(8), gettytab(8).

BUGS

Ideally, the speed tables would be read from a file, not compiled into *getty*.

NAME

gettytab - defining speed tables for getty

SYNOPSIS

/usr/src/cmd/gettytab.c
/usr/src/cmd/getty.object
/usr/src/cmd/getty.mk

DESCRIPTION

When *getty*(8) is called, it looks at **/etc/inittab**. Each line in **/etc/inittab** specifies one of several "speed tables" within *getty*. Based on the speed table argument given in **/etc/inittab,** *getty* sets the "initial" mode of a serial port, prints the "login: " message, reads the response, and sets the "final" mode of the serial port. *Getty* then terminates by executing the *login*(1) program.

Gettytab.c allows you to define additional speed tables, over and above those already defined in *getty*.

To define new speed tables, add speed-table entries to the file */usr/src/cmd/gettytab.c*, and recompile and install the file along with *getty.object* as prescribed within the *getty.mk* file. Choose a speed table name other than those already known to *getty*. See *getty*(8) for this list. Note that "a" and "b" are already known. Attempts to redefine a known speed table will fail.

To test the changed *getty* before installing it as **/etc/getty**, copy the new *getty* to a user directory and make an entry in **/etc/inittab** that refers to the new *getty*. For example, in **/etc/inittab** change the line

    2:11:c:/etc/getty tty11 b

to

    2:11:c:/usr/you/getty tty11 c

for serial port 11 and speed table "c". Only one entry in **/etc/inittab** may refer to serial port 11. To activate the new *getty* type

    init 2

When the user on serial port 11 logs off, the new *getty* will be executed.

NOTES

This is a Plexus command. It is not part of standard SYSTEM III.

SEE ALSO

getty(8), init(8), inittab(5), tty(4).

NAME

init - process control initialization

SYNOPSIS

/etc/init [state]

DESCRIPTION

Init is invoked inside UNIX as the last step in the boot procedure. It is process number one, and is the ancestor of every other process in the system. As such, it can be used to control the process structure of the system. If *init* is invoked with an argument by the super-user, it will cause a change in state of process one.

Init has 9 states, 1 through 9; it is invoked by the system in state 1, and it performs the same functions on entering each state. When a state is entered, *init* reads the file **/etc/inittab**. Lines in this file have the format:

state:id:flags:command

All lines in which the state field matches *init*'s current state are recognized. If a process is active under the same two character *id* as a recognized line, it may be terminated (signal 15), killed (signal 9), or both by including the *flags* **t** and **k** in the order desired. The signal is sent to all processes in the process group associated with the *id*. The *command* field is saved for later execution.

On first invocation by Sys3, *init* sees if *autoboot* is enabled. If it is, *init* puts the system in state 8, the autoboot state.

After reading **/etc/inittab** and signaling running processes as required, but before invoking any processes under the new state, **/etc/rc** is invoked with three arguments. This command file performs housekeeping such as removing temporary files, mounting file systems, and starting daemons. The three arguments are the current state, the number of times this state has been entered previously, and the prior state. *Init* will also execute **/etc/rc** at the request of the operating system (e.g., when recovering from power failure). In this last case, the first argument has an **x** appended to it.

When **/etc/rc** has finished executing, *init* invokes all *commands* waiting to be executed. (A *command* is waiting to be executed if there is no process currently running that has the same *id* as the command.) The *flag* **c** (continuous) requires the *command* to be continuously reinvoked whenever the process with that *id* dies. The *flag* **o** (off) causes the *command* to be ignored. This is useful for turning lines off without extensive editing. Otherwise, the *command* is invoked a maximum of one time in the current state.

Init invokes the *command* field read from **/etc/inittab** by opening / for reading and writing on file descriptors 0, 1, and 2, resetting all signals to system default, setting up a new process group (*setpgrp*(2)), and *exec*ing:

/bin/sh -c exec *command*

DIAGNOSTICS

When *init* can do nothing else because of a missing **/etc/inittab** or when it has no children left, it will try to execute a shell on **/dev/console**. When the problem has been fixed, it is necessary to change states, and terminate the shell.

BUGS

Init does not complain if the state-id pairs in **/etc/inittab** are not unique. For any given pair, the last one in the file is valid.

FILES

/etc/inittab
/etc/rc
/bin/sh

/dev/console

SEE ALSO

login(1), sh(1), exec(2), setpgrp(2), inittab(5), getty(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, ., /, and upper- and lower-case letters. 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 4,096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but broken in 4,096 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 *output key* bits in the result.

Makekey is intended for programs that perform encryption (e.g., *ed*(1) and *crypt*(1)). Usually, its input and output will be pipes.

SEE ALSO

crypt(1), ed(1), passwd(5).

NAME

mk - how to remake the system and commands

DESCRIPTION

All source for UNIX is in a source tree distributed in the directory **/usr/src**. This includes source for the operating system, libraries, commands, miscellaneous files necessary to the running system, and procedures to create everything from this source.

The top level consists of the directories **cmd, lib, uts, head,** and **stand** as well as commands to remake each of these "directories". These commands are named *:mk*, which remakes everything, and *:mk***dir** where **dir** is the directory to be recreated. Each recreation command will make all or part of the piec; over which it has control. *:mk* will run each of these commands and thus recreate the whole system.

The **lib** directory contains libraries used when loading user programs. The largest and most important of these is the C library. All libraries are in sub-directories and are created by a makefile or runcom. A runcom is a Shell command procedure used specifically to remake a piece of the system. *:mklib* will rebuild the libraries that are given as arguments. The argument ***** will cause it to remake all libraries.

The **head** directory contains the header files, usually found in **/usr/include** on the running system. *:mkhead* will install those header files that are given as arguments. The argument ***** will cause it to install all header files.

The **uts** directory contains the source for the UNIX operating system. *:mkuts* (no arguments) invokes a series of makefiles that will recreate the operating system.

The **stand** directory contains stand-alone commands and boot programs. *:mkstand* will rebuild and install these programs.

The **cmd** directory contains files and directories. *:mkcmd* transforms source into a command based upon its suffix (**.l**, **.y**, **.c**, **.s**, **.sh**), or its makefile (see *make*(1)) or runcom. A directory is assumed to have a makefile or a runcom that will take care of creating everything associated with that directory and its sub-directories. Makefiles and runcoms are named *command***.mk** and *command***.rc** respectively.

:mkcmd will recreate commands based upon a makefile or runcom if one of them exists; alternatively commands are recreated in a standard way based on the suffix of the source file. All commands requiring more than one file of source are grouped in sub-directories, and must have a makefile or a runcom. C programs (**.c**) are compiled by the C compiler and loaded stripped with shared text. Assembly language programs (**.s**) are run through the preprocessor of *cc*(−Pflag) and then assembled. The file ldflags in **/usr/src** override the default flags to the loader, *ld*. This way, some commands can be loaded with the **-i** flag. Yacc programs (**.y**) and lex programs (**.l**) are processed by *yacc*(1) and *lex*(1) respectively before C compilation. Shell programs (**.sh**) are copied to create the command. Each of these operations leaves a command in **./cmd** which is then installed by using **/etc/install**.

The arguments to *:mkcmd* are either command names, or subsystem names. The subsystems distributed with UNIX are: **acct**, **graf**, **rje**, **sccs**, and **text**. Prefacing the *:mkcmd* instruction with an assignment to the Shell variable **$ARGS** will cause the indicated components of the subsystem to be rebuilt.

The entire **sccs** subsystem can be rebuilt by:

    /usr/src/:mkcmd sccs

while the *delta* component of **sccs** can be rebuilt by:

    ARGS="delta" /usr/src/:mkcmd sccs

The *log* command, which is a part of the **stat** package, which is itself a part of the **graf** package, can be rebuilt by:

ARGS="stat log" /usr/src/:mkcmd graf

The argument ***** will cause all commands and subsystems to be rebuilt.

Makefiles, both in **./cmd** and in sub-directories, have a standard format. In particular *:mkcmd* depends on there being entries for *install* and *clobber*. *Install* should cause everything over which the makefile has jurisdiction to be made and installed by **/etc/install**. *Clobber* should cause a complete cleanup of all unnecessary files resulting from the previous invocation.

The file **/etc/places** defines several source and destination directories.

Most of the runcoms in **./cmd** (as opposed to sub-directories) relate in particular to a need for separated instruction and data (I and D) space.

In the past, dependency on the C library routine *ctime*(3C) was also important. *Ctime* had to be modified for all systems located outside of the Pacific time zone, and all commands that referenced it had to be recompiled. *Ctime* has been rewritten to check the environment (see *environ*(7)) for the time zone. This results in time zone conversions possible on a per-process basis. **/etc/profile** sets the initial environment for each user, and **/etc/rc** sets it for certain system daemons. These two programs are the only ones which must be modified outside of the Pacific time zone.

An effort has been made to separate the creation of a command from source, and its installation on the running system. The command **/etc/install** is used by *:mkcmd* and most makefiles to install commands in the proper place on the running system. The use of install allows maximum flexibility in the administration of the system. Install makes very few assumptions about where a command is located, who owns it, and what modes are in effect. All assumptions may be overridden on invocation of the command, or more permanently by redefining a few variables in install. The object is to install a new version of a command in the same place, with the same attributes as the prior version.

In addition, the use of a separate command to perform installation allows for the creation of test systems in other than standard places, easy movement of commands to balance load, and independent maintenance of makefiles. The minimization of makefiles in most cases, and the site independence of the others should greatly reduce the necessary maintenance, and allow makefiles to be considered part of the standard source.

SEE ALSO

install(1M), make(1).

NAME

rc - system initialization shell script

SYNOPSIS

/etc/rc

DESCRIPTION

The **/etc/rc** file is executed by *init*(8) whenever the *init* state is changed.

SEE ALSO

init(8).

NAME

rje - RJE (Remote Job Entry) to IBM

SYNOPSIS

/usr/rje/rjeinit
/usr/rje/rjehalt

DESCRIPTION

RJE is the communal name for a collection of programs and a file organization that allows a UNIX system, equipped with an ICP driver, and associated Virtual Protocol Machine (VPM) software, to communicate with IBM's Job Entry Subsystems by mimicking an IBM 360 remote multileaving work station.

Implementation.

RJE is initiated by the command *rjeinit* and is terminated gracefully by the command *rjehalt*. While active, RJE runs in the background and requires no human supervision. It quietly transmits, to the IBM system, jobs that have been queued by the *send*(1C) command, and operator requests that have been entered by the *rjestat*(1C) command. It receives, from the IBM system, print and punch data sets and message output. It enters the data sets into the proper UNIX directory and notifies the appropriate user of their arrival. It scans the message output to maintain a record on each of its jobs. It also makes these messages available for public inspection, so that *rjestat*(1C), in particular, may extract responses.

Unless otherwise specified, all files and commands described below reside in directory **/usr/rje** (first exceptions: *send* and *rjestat*).

There are two sources of data to be transmitted by RJE from UNIX to an IBM System/370. In both cases, the data is organized as files in the **/usr/rje/squeue** directory. The first are files named **co*** which are created by the enquiry command *rjestat*(1C). The second source, containing the bulk of the data, are files named **rd*** or **sq*** which have been created by *send* and queued by the program *rjeqer*. On completion of processing *send* invokes *rjeqer*. *Rjeqer* and *rjestat* inform the program *rjexmit* that a file has been queued via the file **joblog**. Upon successful transmission of the data to the IBM machine, *rjexmit* removes the queued file. As files are transmitted and received, the program *rjedisp* writes an entry containing the date, time, file name, logname, and number of records in the file **acctlog**, if it exists. This file can be used for local logging or accounting information, but is not used elsewhere by RJE. The use of this information is up to the RJE administrator.

Each time *rjeinit* is invoked, the **joblog** file is truncated and recreated from the contents of the **/usr/rje/squeue** directory. During this time, *rjeinit* prevents simultaneous updating of the **joblog** file.

Output from the IBM system is classified as either a print data set, a punch data set, or message output. Print output is converted to an ASCII text file, with standard tabs. Form feeds are suppressed, but the last line of each page is distinguished by the presence of an extraneous trailing space. Punch output is converted to *pnch*(5) format. This classification and both conversions occur as the output is received. Files are moved or copied into the appropriate user's directory and assigned the name **prnt*** or **pnch***, respectively, or placed into user directories under user-specified names, or used as input to programs to be automatically executed, as specified by the user. This process is driven by the "usr=..." specification. RJE retains ownership of these files and permits read-only access to them. Message output is digested by RJE immediately and is not retained.

A record is maintained for each job that passes through RJE. Identifying information is extracted contextually from files transmitted to and received from the IBM system. This information is stored and used by the *rjedisp* program for IBM job acknowledgements and delivery of output files.

The IBM system automatically returns an acknowledgement message for each job it receives. Other status messages are returned in response to enquiries entered by users. All messages received by RJE are appended to the **resp** file. The **resp** file is automatically truncated when it reaches 70,000 bytes. Each enquiry is preceded and followed by an identification card image of the form "$UX<*process id*>". The IBM system will echo this back as an illegal command. The appearance of process ids in the response stream permits responses to be passed on to the proper users.

While it is active, RJE occupies at least the three process slots that are appropriated by *rjeinit*. These slots are used to run *rjexmit*, the transmitter, *rjerecv*, the receiver, and *rjedisp*, the dispatcher. These three processes are connected by pipes. The function of each is as follows:

rjexmit Cycles repetitively, looking for data to transmit to the IBM system. After transmission, *rjexmit* passes an event notice to *rjedisp*. If *rjexmit* encounters a **stop** file, (created by *rjehalt*), it exits normally. In the case of error termination, *rjexmit* reboots RJE by executing *rjeinit*.

rjerecv Cycles repetitively, looking for data returning from the IBM machine. Upon receipt of data, *rjerecv* notifies either *rjexmit* or *rjedisp* of the event (transfer information is sometimes passed to *rjexmit*). *Rjerecv* exits normally at the first appropriate moment when it encounters the file **stop**, or exits reluctantly when it encounters a run of errors.

rjedisp Follows up event notices by directing output files, updating records, and notifying users. *Rjedisp* references the system files **/etc/passwd** and **/etc/utmp** to correlate user names, numeric ids, and terminals. Termination of *rjerecv* causes *rjedisp* to exit also.

Most RJE files and directories are protected from unauthorized tampering. The exception is the **spool** directory. It is used by *send*(1C) to create temporary files in the correct file system. *Rjeqer* and *rjestat*(1C), the user's interfaces to RJE, operate in *setuid* mode to contribute the necessary permission modes.

Administration.

Some minimal oversight of each RJE subsystem is required. The RJE mailbox should be inspected and cleaned out periodically. The **job** directory should also be checked. The only files placed there are output files whose destination file systems are out of space. Users should be given a short period of time (say, a day or two), and then these files should be removed.

The configuration table **/usr/rje/lines** is accessed by all components of RJE. Each line of the table (maximum of 8) defines an RJE connection. Its seven columns may be labeled *host*, *system*, *directory*, *prefix*, *device*, *peripherals* and *parameters*. These columns are described as follows:

host
The name of a remote IBM computer (e.g., **A B C**). This string can be up to 5 characters.

system
The name of a UNIX system. This name should be the same as the system name from *uname*(1).

directory
This is the directory name of the servicing RJE subsystem (e.g., **/usr/rje1**).

prefix
This is the string prefixed (redundantly) to several crucial files and programs in **directory** (e.g., **rje1**, **rje2**, **rje3**).

device
This is the name of the controlling VPM device, with **/dev/** excised.

peripherals

This field contains information on the logical devices (readers, printers, punches) used by RJE. Each subfield is separated by :, and is described as follows:

(1) Number of logical readers.
(2) Number of logical printers.
(3) Number of logical punches.

Note: the number of peripherals specified for an RJE subsystem **must** agree with the number of peripherals which have been described on the remote machine for that line.

parameters

This field contains information on the type of connection to make. Each subfield is separated by :. Any or all fields may be omitted; however, the fields are positional. All but trailing delimiters must be present. For example, in

1200:512:::9-555-1212

subfields 3 and 4 are missing, but the delimiters are present. Each subfield is defined as follows:

(1) **space**

This subfield specifies the amount of space (S) in blocks that RJE tries to maintain on file systems it touches. The default is 0 blocks. *Send* will not submit jobs and *rjeinit* issues a warning when less than 1.5S blocks are available; *rjerecv* stops accepting output from the host when the capacity falls to S blocks; RJE becomes dormant, until conditions improve. If the space on the file system specified by the user on the "usr=" card would be depleted to a point below S, the file will be put in the **job** subdirectory of the connection's home directory, rather than in the place that the user requested.

(2) **size**

This subfield specifies the size in blocks of the largest file that can be accepted from the host without truncation taking place. The default is no truncation.

(3) **badjobs**

This subfield specifies what to do with undeliverable returning jobs. If an output file is undeliverable for any reason other than file system space limitations (e.g., missing or invalid "usr=" card) and this subfield contains the letter **y**, the output will be retained in the **job** subdirectory of the home directory, and login **rje** is notified. If this subfield contains an **n** or has any other value, undeliverable output will be discarded. The default is **n**.

(4) **console**

This subfield specifies the status of the interactive status terminal for this line. If the subfield contains an **i**, all console status facilities are inhibited (e.g., *rjestat*(1C) will not behave like a status terminal). In all cases, the normal non-interactive uses of *rjestat*(1C) will continue to function. The default is **y**.

Sign-on is controlled by the existence of a **signon** file in the home directory. If this file is present, its contents are sent as a sign-on message to the host system. If this file does not exist, a blank card is sent. Sign-off is controlled in the same way, except that the **signoff** file is sent by *rjehalt* if it exists. If the **signoff** file does not exist, a "**/*signoff**" card is sent. These files should be ASCII text and no more than 80 characters.

Send(1C) and *rjestat*(1C) select an available connection by indexing on the **host** field of the configuration table. RJE programs index on the **prefix** field. A subordinate directory, **sque**, exists in **/usr/rje** for use by *rjedisp* and *shqer* programs. This directory holds those output files that have been designated as standard input to some executable file. This designation is done via the "usr=..." specification. *Rjedisp* places the output files here and updates the file **log** to specify the order of execution, arguments to be passed, etc. *Shqer* executes the appropriate files.

All RJE programs are shared text; therefore, if more than one RJE is to be run on a given UNIX system, simply link (via *ln*(1)) RJE2 program names to RJE names in **/usr**.

SEE ALSO

rjestat(1C), send(1C), vpm(4), pnch(5), mk(8).
UNIX Remote Job Entry User's Guide by K. A. Kelleman.
UNIX Remote Job Entry Administrative Guide by M. J. Fitton.
Setting Up UNIX.

DIAGNOSTICS

Rjeinit provides brief error messages describing obstacles encountered while bringing up RJE. They can best be understood in the context of the RJE source code. The most frequently occurring one is "cannot open /dev/vpm?". This may occur if the VPM script has not been started, or if another process already has the VPM device open.

Once RJE has been started, users should assist in monitoring its performance, and should notify operations personnel of any perceived need for remedial action. *Rjestat*(1C) will aid in diagnosing the current state of RJE. It can detect, with some reliability, when the far end of the communications line has gone dead, and will report in this case that the host computer is not responding to RJE. It will also attempt to reboot RJE if it detects a prolonged period of inactivity on the ICP.

NAME

sar - system activity report package

DESCRIPTION

Sar is the first (tentative) piece of an overall UNIX measurement and statistics package; the data that are collected and the output formats are not yet final.

The operating system contains a number of counters that are incremented as various system actions occur. These include several time counters (that are incremented each 50th of a second depending on the CPU mode), I/O activity counters, switching and system-call counters, and file-access counters. The system activity package writes system activity parameters periodically on a binary file. It also generates a daily system activity report that covers the prime period (from 8:00 to 18:00).

The data collection and report generation are controlled by entries in **crontab** (see *cron*(1M)). The data collection program is normally activated every hour on the hour; the report generation once a day.

Every time the system is booted, a special record is written to the daily data file, since all the system activity counters restart from zero at that time. This process is done while executing **/etc/rc** see (*init*(8)) during UNIX initialization. It produces an entry on the daily report showing the restart time.

The daily reports are deposited in **/usr/adm/sa/sar***dd* where *dd* are digits representing the day of the month. A report can be printed (e.g., **cat /usr/adm/sa/sar05**) any time before it is removed the following week.

The structure of the binary daily data file is:

```
struct sa {
        struct sysinfo si; /* defined in /usr/include/sys/sysinfo.h */
        long d0;           /* number of reads and writes of disk 0 */
        long d1;           /* number of reads and writes of disk 1 */
        long d2;           /* number of reads and writes of disk 2 */
        long ts;           /* time stamp in time_t format */
};
```

FILES

/usr/adm/sa/sa*dd*	daily data file
/usr/adm/sa/sar*dd*	daily report file
/tmp/sa.adrfl	address file

NAME

shutdown - terminate all processing

SYNOPSIS

/etc/shutdown

DESCRIPTION

Shutdown is part of the UNIX operation procedures. Its primary function is to terminate all currently running processes in an orderly and cautious manner. The procedure is designed to interact with the operator (i.e., the person who invoked *shutdown*). *Shutdown* may instruct the operator to perform some specific tasks, or to supply certain responses before execution can resume. *Shutdown* should be run from the system console by **root**.

Shutdown goes through the following steps:

- All users logged on the system are notified to log off the system by a broadcasted message. The operator may display his/her own message at this time. Otherwise, the standard file save message is displayed.
- If the operator wishes to run the file-save procedure, *shutdown* unmounts all file systems.
- All file systems' super blocks are updated before the system is to be stopped (see *sync*(1M)). This must be done before re-booting the system, to insure file system integrity.

Shutdown does not terminate processes associated with the operator's terminal. The most common error diagnostic that will occur is *device busy*. This diagnostic happens when a particular file system could not be unmounted. See *umount*(1M).

SEE ALSO

sync(1M), umount(1M).