# PRO/VENIX™

UNIX™
for

Professional™

Programmer Reference Manual

digital software

# PRO/VENIX™

## for the Professional

## Programmer Reference Manual

Developed by:

VenturCom, Inc.
215 First Street
Cambridge, MA 02142

Digital Equipment Corporation
Maynard, MA 01754

First Printing

The software described in this manual is distributed as part of Digital Equipment Corporation's Digital Classified Software (DCS) Program. This program enables software developers to submit their software products to Digital for testing according to Digital quality standards for third party software. This software product has met the DCS standard specified in the software product description (SPD) for this product. You should refer to the SPD for information about these standards, the hardware and software required to run this product, and warranties (if any warranty is available).

The software described in this manual is furnished under a license and may only be used or copied in accordance with the terms of that license. This manual is reproduced with the permission of VenturCom, Inc.



Except as may be stated in the SPD for this product, no responsibility is assumed by Digital or its affiliated companies for use or reliability of this software, or for errors in this manual or in the software. Additional support and/or warranty services may be available from the developer of this software product. Digital has no connection with, and assumes no responsibility or liabilities in connection with these services.

This manual is subject to change without notice and does not constitute a commitment by Digital.

VENIX is a trademark of VenturCom, Inc.
UNIX is a trademark of AT&T Technology, Inc.

The following are trademarks of Digital Equipment Corporation:

| | | | |
|---|---|---|---|
| DEC | DECwriter | Professional | VAX |
| DECmate | DIBOL | Rainbow | VMS |
| DECnet | MASSBUS | RSTS | VT |
| DECsystem-10 | PDP | RSX | Work Processor |
| DECSYSTEM-20 | P/OS | UNIBUS | digital |
| DECUS | | | |

## The PRO/VENIX† Documentation Set

The PRO/VENIX documentation set consists of the following manuals:

*PRO/VENIX Installation and System Manager's Guide*

The set up and maintenance of PRO/VENIX are described in the installation sections. Other articles explain the UNIX-to-UNIX‡ communications systems. The "System Maintenance Reference Manual" contains reference pages for devices and system maintenance procedures (sections (7) and (8)).

*PRO/VENIX User Guide*

The *User Guide* contains tutorials for newcomers to PRO/VENIX, covering basic use of the system, the editor **vi** and use of the command language interpreters.

*PRO/VENIX Document Processing Guide*

The line and screen editors and **nroff**-related text formatting utilities are described in the Document Processing Guide. Topics include: line editor **ed**, and stream editor **sed**; the text formatter **nroff**; the **nroff**-preprocessors **tbl** and **neqn**.

*PRO/VENIX Programming Guide*

The chapters in the *Programming Guide* explicate the different programming languages for VENIX.

---

† VENIX is a trademark of VenturCom, Inc.

‡ UNIX is a trademark of Bell Laboratories.

*PRO/VENIX Support Tools Guide*

This guide includes tools for programming, such as the compiler-writing languages Yacc and Lex, the M4 Macro processor, the program development utility Make, and the desk calculator programs DC and BC.

*PRO/VENIX User Reference Manual*

This is a complete and concise reference for the PRO/VENIX system. This volume contains write-ups on all PRO/VENIX commands.

*PRO/VENIX Progammer Reference Manual*

The reference pages in this volume include system calls, library functions, file formats, miscellaneous functions and games.

# Table of Contents

## 2. SYSTEM CALLS

# Table of Contents

(SYSTEM CALLS continued)



**3. SUBROUTINES**

# Table of Contents

(SUBROUTINES continued)

# Table of Contents

(SUBROUTINES continued)



## 4. FILE FORMATS AND CONVENTIONS



## 5. MISCELLANEOUS FACILITIES

# Table of Contents

## 6. GAMES

# Contents

2

## NAME

intro, errno — introduction to system calls and error numbers

## SYNOPSIS

**#include <errno.h>**

## DESCRIPTION

Section 2 of this manual lists all the entries into the system. Use of many of these calls is discussed in the chapter "VENIX Programming" in the *Programming Guide*. You might also consider use of the standard I/O library, described in section 3 of this manual as well as in "VENIX Programming," for easier use of many of the I/O functions. Be careful not to mix system-level I/O calls (such as **open()** and **close()**) with standard I/O calls (such as **fopen()**and **fclose()**).

Most of these calls have an error return. An error condition is indicated by an otherwise impossible returned value. Almost always this is −1; the individual sections specify the details. An error number is also made available in the external variable **errno**. **errno** is not cleared on successful calls, so it should be tested only after an error has occurred.

There is a table of messages associated with each error, and a routine for printing the message; see **perror(3)**. The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in **<errno.h>**, and the messages available using **perror**.

Errors which have an '*' preceding the number may have an accompanying system error message on the main console device, since they represent conditions which the system administrator should be aware of. See "VENIX Maintenance" in the *Installation and System Manager's Guide* for a listing of system error messages.

0 — Error 0 Unused

1 **EPERM** — Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 **ENOENT** — No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 **ESRCH** — No such process
The process whose number was given to **signal**, **suspend**, or **ptrace** does not exist, or is already dead.

4 **EINTR** — Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

*5 **EIO** — I/O error
Some physical I/O error occurred during a **read** or **write**. This error may in some cases occur on a call following the one to which it actually applies.

6 **ENXIO** — No such device or address
I/O on a special file refers to a subdevice that does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialed in or no disk pack is loaded on a drive.

7 **E2BIG** — Arg list too long
An argument list longer than 2048 bytes is presented to **exec**.

8 **ENOEXEC** — Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see **a.out**(4).

9 **EBADF** — Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file that is open only for writing (resp. reading).

10 **ECHILD** — No children
**wait** and the process has no living or unwaited-for children.

11 **EAGAIN** — No more processes
In a **fork**, the system's process table is full or the user is not allowed to create any more processes.

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

13 **EACCES** — Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 **EFAULT** — Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

15 **ENOTBLK** — Block device required
A plain file was mentioned where a block device was required, e.g. in **mount**.

16 **EBUSY** — Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment).

17 **EEXIST** — File exists
An existing file was mentioned in an inappropriate context, e.g. **link**.

18 **EXDEV** — Cross-device link
A link to a file on another device was attempted.

19 **ENODEV** — No such device
An attempt was made to apply an inappropriate system call to a device, e.g. read a write-only device.

20 **ENOTDIR** — Not a directory
A non-directory was specified where a directory is required, for example in a pathname or as an argument to **chdir**.

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

22 **EINVAL** — Invalid argument
Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in **signal**, reading or writing a file for which **seek** has generated a negative pointer. Also set by math functions, see **intro**(3).

*23 **ENFILE** — File table overflow
The system's table of open files is full, and temporarily no more **opens** can be accepted.

24 **EMFILE** — Too many open files
Customary configuration limit is 15 per process.

25 **ENOTTY** — Not a typewriter
The file mentioned in **ioctl** is not a terminal or one of the other devices to which these calls apply.

26 **ETXTBSY** — Text file busy
An attempt to execute a pure-procedure program that is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 **EFBIG** — File too large
The size of a file exceeded the maximum (about 16 mbytes).

*28 **ENOSPC** — No space left on device
During a **write** to an ordinary file, there is no free space left on the device.

29 **ESPIPE** — Illegal seek
An **lseek** was issued to a pipe. This error should also be issued for other non-seekable devices.

30 **EROFS** — Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 **EMLINK** — Too many links
An attempt to make more than 127 links to a file.

32 **EPIPE** — Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 **EDOM** — Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 **ERANGE** — Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

## SEE ALSO

intro(3)

## ASSEMBLER

Assembler interface is given for both PDP-11 and 8086 processors.

PDP-11:
The assembler symbols are defined in **/usr/include/sys.s**. Return values appear in registers R0 and R1; it is unwise to count on these registers being preserved when no value is expected. An erroneous call is always indicated by turning on the C-bit of the condition codes. The error number is returned in R0. The presence of an error is most easily tested by the instructions **bes** and **bec** ('branch on error set (or clear)'). These are synonyms for the **bcs** and **bcc** instructions.

8086: Return values appear in registers AX, DX and CX; it is unwise to count on these registers being preserved when no value is expected. An erroneous call is always indicated by an error number in CX. The presence of an error is most easily tested by the instruction **JCXZ** ("jmp CX zero").

Cross-Reference to VENIX System Calls

| | | | |
|---|---|---|---|
| **access** | **ACCESS** | **link** | **LINK** |
| **aiowait** | **AIOWAIT** | **lock** | **LOCK** |
| **alarm** | **ALARM** | **lseek** | **LSEEK** |
| **break** | **BRK** | **mknod** | **MKNOD** |
| **brk** | **BRK** | **mount** | **MOUNT** |
| **chdir** | **CHDIR** | **nice** | **NICE** |
| **chmod** | **CHMOD** | **open** | **OPEN** |
| **chown** | **CHOWN** | **pause** | **PAUSE** |
| **chroot** | **CHDIR** | **phys** | **PHYS** |
| **close** | **CLOSE** | **pipe** | **PIPE** |
| **cmap** | **CMAP** | **profil** | **PROFIL** |
| **creat** | **CREAT** | **ptrace** | **PTRACE** |
| **dup** | **DUP** | **read** | **READ** |
| **dup2** | **DUP** | **sbrk** | **BRK** |
| **environ** | **EXEC** | **sdata** | **SDATA** |
| **errno** | **INTRO** | **semclear** | **SEMSET** |
| **exec** | **EXEC** | **semset** | **SEMSET** |
| **exece** | **EXEC** | **semtest** | **SEMSET** |
| **execl** | **EXEC** | **semtset** | **SEMSET** |
| **execle** | **EXEC** | **setgid** | **SETUID** |
| **execlp** | **EXEC** | **setuid** | **SETUID** |
| **execv** | **EXEC** | **signal** | **SIGNAL** |
| **execve** | **EXEC** | **stat** | **STAT** |
| **execvp** | **EXEC** | **stime** | **STIME** |
| **exit** | **EXIT** | **stty** | **IOCTL** |
| **fork** | **FORK** | **suspend** | **SUSPEND** |
| **fstat** | **STAT** | **sync** | **SYNC** |
| **ftime** | **TIME** | **tell** | **LSEEK** |
| **getegid** | **GETUID** | **time** | **TIME** |
| **geteuid** | **GETUID** | **times** | **TIMES** |
| **getgid** | **GETUID** | **umask** | **UMASK** |
| **getpid** | **GETPID** | **umount** | **MOUNT** |
| **getuid** | **GETUID** | **unlink** | **UNLINK** |
| **gtty** | **IOCTL** | **utime** | **UTIME** |
| **indir** | **INDIR** | **wait** | **WAIT** |
| **intro** | **INTRO** | **write** | **WRITE** |
| **ioctl** | **IOCTL** | | |
| **kill** | **KILL** | | |
| **libmon** | **LIBMON** | | |

## NAME

access — determine accessibility of file

## SYNOPSIS

**access(name, mode)**
**char *name;**

## DESCRIPTION

**access** checks the given file *name* for accessibility according to *mode*, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if *name* cannot be found or if any of the desired access modes would not be granted. On disallowed accesses $-1$ is returned and the error code is in **errno**. 0 is returned from successful tests.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by **access**, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but **exec** will fail unless it is in proper format.

## SEE ALSO

access(1), stat(2)

## ASSEMBLER

(access = 33.)

PDP-11:
    sys acess; name; mode

8086: BX = 33
    AX = name
    DX = mode
    int 0Xf1

## NAME

aiowait — wait on asynchronous I/O

## SYNOPSIS

**aiowait(fd, level)**

## DESCRIPTION

**aiowait** causes the calling process to go to sleep until the outstanding I/O requests by the process to the device referred to by the file descriptor *fd* are less then or equal to *level*. The number of outstanding requests is returned. If *level* is negative, then only the number of outstanding requests is returned.

*fd* is the file descriptor returned by a previous **open** of the asynchronous version of a DMA device, such as a disk or A/D device.

Since asynchronous I/O is serviced in the order requested, the user can know when a given request has been completed.

**aiowait** is implemented by a call to **ioctl**(2) with the **aiocwait** command.

## SEE ALSO

ioctl(2), async(7)

## DIAGNOSTICS

A $-1$ is returned if the file descriptor is unknown or not a special character file opened for asynchronous I/O.

## NOTES

Asynchronous I/O is non-portable to standard UNIX.

## NAME

alarm — schedule signal after specified time

## SYNOPSIS

**alarm(time)**

## DESCRIPTION

**alarm** causes signal SIGALRM (see **signal**(2)) to be sent to the invoking process in a specified time given by the argument. Unless caught or ignored, the signal terminates the process.

If *time* is greater than zero, the alarm will be measured in seconds. Successive calls of positive alarm values will not be stacked; an alarm will be sent only at the time indicated by the most recent call. The return value will be the amount of time previously remaining on the clock. A call with value zero will cancel the last positive-valued alarm. The longest specifiable positive *time* value is 32767.

If *time* is a negative value the alarm will be measured in clock-ticks, i.e. 1/60th of a second, and equal to the absolute value of *time*. (Note: not all machines have a clock running precisely 60Hz, thus the alarm scheduling granularity may be larger than 1/60th of a second.) Successive calls of negative alarm values will not be stacked; the alarm will be sent at the time indicated by the most recent call. Unlike positive alarms, this type of alarm can not be cancelled, and the return value for this type of call is always zero. The longest specifiable alarm in clock-ticks is 32768.

Note that **alarm** calls given in seconds (positive or zero *time*) and those given in clock-ticks (negative *time*) are handled totally apart, and can almost be considered separate system calls.

Because of the resolution of the respective clocks, alarms given in seconds may be up to one second early, and alarms given in clock-ticks may be up to one clock-tick early. Because of scheduling delays, resumption of execution when the signal is caught may be delayed an arbitrary amount.

## SEE ALSO

pause(2), signal(2), sleep(3)

## NOTES

Clock-tick alarms are not portable to standard UNIX, and VENIX does not support regular alarms for longer than 32767 seconds.

**ASSEMBLER**

(alarm = 27.)

PDP-11:

R0 = time
sys alarm
R0 = previous amount or zero

8086: BX = 27
AX = time
int 0Xf1
AX = previous amount or zero

## NAME

brk, sbrk, break — change core allocation

## SYNOPSIS

**char *brk(addr)**

**char *sbrk(incr)**

## DESCRIPTION

**brk** sets the system's idea of the lowest location not used by the program (called the break) to *addr*. Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function **sbrk**, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via **exec**(2) the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use these calls.

## SEE ALSO

exec(2), malloc(3), end(3)

## DIAGNOSTICS

Zero is returned if the break could be set; −1 if the program requests more memory than the system limit.

## ASSEMBLER

(break = 17.)

PDP-11:

```
sys break; addr
```

8086:

```
BX = 17
AX = addr
int 0Xf1
```

**break** performs the function of **brk**. The name of the routine differs from that in C for historical reasons.

## NAME

chdir, chroot — change default directory

## SYNOPSIS

**chdir(dirname)**
**char *dirname;**

**chroot(dirname)**
**char *dirname;**

## DESCRIPTION

*dirname* is the address of the pathname of a directory, terminated by a null byte. **chdir** causes this directory to become the current working directory, the starting point for pathnames not beginning with '/'.

**chroot** sets the root directory, the starting point for pathnames beginning with '/'. (Note that this applies to the calling process alone, not to the complete system.)

## SEE ALSO

cd(1)

## DIAGNOSTICS

Zero is returned if the directory is changed; −1 is returned if the given name is not that of a directory or is not searchable.

## ASSEMBLER

(chdir = 12.)

PDP-11:
sys chdir; dirname

8086: BX = 12
AX = dirname
int 0Xf1

(chroot = 61.)

PDP-11:
sys chroot; dirname

8086: BX = 61
AX = dirname
int 0Xf1

## NAME

chmod — change mode of file

## SYNOPSIS

**chmod(name, mode)**
**char *name;**

## DESCRIPTION

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode*. Modes are constructed by OR'ing together some combination of the following:

04000 set user ID on execution
02000 set group ID on execution
01000 save text image after execution
00400 read by owner
00200 write by owner
00100 execute (search on directory) by owner
00070 read, write, execute (search) by group
00007 read, write, execute (search) by others

If an executable file is set up for sharing (**−n** or **−i** option of **ld**(1)) then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worthwhile for heavily used commands. This is only in effect for files residing on the root file system. A program with this bit set should not be removed or replaced if it has been executed at all since the last system boot-up (as an unreferenced i-node results). The correct procedure is to remove/replace the file before it has been executed following a boot-up.

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

## SEE ALSO

chmod(1)

## DIAGNOSTIC

Zero is returned if the mode is changed; −1 is returned if *name* cannot be found or if current user is neither the owner of the file nor the super-user.

**ASSEMBLER**

(chmod = 15.)

PDP-11:

```
sys chmod; name; mode
```

8086:

```
BX = 15
AX = name
DX = mode
int 0Xf1
```

## NAME

chown — change owner and group of a file

## SYNOPSIS

**chown(name, owner, group)**
**char *name;**

## DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its *owner* and *group* changed as specified. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (non-existent) file-space accounting procedures.

## SEE ALSO

chown(1), passwd(4)

## DIAGNOSTICS

Zero is returned if the owner is changed; −1 is returned on illegal owner changes.

## ASSEMBLER

(chown = 16.)

PDP-11:

```
sys chown; name; owner; group
```

8086:

```
BX = 16
AX = name
DX = owner
CX = group
int 0Xf1
```

## NAME

close — close a file

## SYNOPSIS

**close(fildes)**

## DESCRIPTION

Given a file descriptor such as returned from an **open**, **creat**, **dup**, or **pipe**(2) call, **close** closes the associated file. A close of all files is automatic on **exit**, but since there is a limit on the number of open files per process, **close** is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain file descriptors may be closed by **exec**(2) (see **ioctl**(2)).

## SEE ALSO

creat(2), dup(2), open(2), pipe(2), exec(2), ioctl(2)

## DIAGNOSTICS

Zero is returned if a file is closed; −1 is returned for an unknown file descriptor.

## ASSEMBLER

(close = 6.)

PDP-11:

```
R0 = fildes
sys close
```

8086:

```
BX = 6
AX = fildes
int 0Xf1
```

## NAME

cmap — remap the code segment of a program

## DESCRIPTION

**cmap** is used to remap a different portion of code into a process' address space. This call is used by a process which is too big to fit into 64kb of address space.

## SEE ALSO

"Code-Mapping Under VENIX" in the *Programming Guide*
ld(1), sdata(2)

## NOTES

Code-mapping is not portable to standard UNIX. This call should not be used directly by the user; the loader is set up to deal with code-mapping when given the **−m** flag.

## ASSEMBLER

(cmap = 62.)

PDP-11:

R0 = offset
sys cmap
R0 = −1 if error, 0 if okay

## NAME

creat — create a new file

## SYNOPSIS

**creat(name, mode)**
**char *name;**

## DESCRIPTION

**creat** creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process' mode mask (see **umask**(2)). Also see **chmod**(2) for the construction of the *mode* argument. The owner ID of the file is set to the process' effective user and group ID.

The file is opened for writing only, and its file descriptor is returned.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a **creat**, an error is returned and the program knows that the name is unusable for the moment. The set-ID and sticky text bits can not be set by this mode; use **chmod**(2) to accomplish this.

## SEE ALSO

write(2), close(2), chmod(2), umask(2)

## DIAGNOSTICS

The value −1 is returned and the file not created if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory. If there are already too many files opened, the file is created but −1 is returned.

## ASSEMBLER

(creat = 8.)

PDP-11:

sys creat; name; mode
R0 = file descriptor

8086: BX = 8
AX = name
DX = mode
int 0Xf1

## NAME

dup, dup2 — duplicate an open file descriptor

## SYNOPSIS

**dup(fildes)**
**int fildes;**

**dup2(fildes, fildes2)**
**int fildes, fildes2;**

## DESCRIPTION

Given a file descriptor returned from an **open**, **pipe**, or **creat**(2) call, **dup** allocates another file descriptor synonymous with the original. The new file descriptor is returned. **dup** always returns the lowest available file descriptor.

In the second form of the call, *fildes* is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than the maximum value allowed for file descriptors (approximately 14). **dup2** causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

## SEE ALSO

creat(2), open(2), close(2), pipe(2)

## DIAGNOSTICS

The value −1 is returned if: the given file descriptor is invalid; there are already too many open files.

## ASSEMBLER

(dup = 41.)

PDP-11:

R0 = file descriptor
R1 = new file descriptor
sys dup
R0 = file descriptor

8086: BX = 41
AX = fildes
DX = new fildes
int 0Xf1

The **dup2** entry is implemented by adding 0100 to *fildes*.

## NAME

execl, execv, execle, execve, execlp, execvp, exec, exece, environ — execute a file

## SYNOPSIS

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

**execv(name, argv)**
**char *name, *argv[ ];**

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

**execve(name, argv, envp);**
**char *name, *argv[ ], *envp[ ];**

**extern char **environ;**

## DESCRIPTION

**exec** in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

Files remain open across **exec**; see **ioctl**(2). Ignored signals remain ignored across these calls, but signals that are caught (see **signal**(2)) are reset to their default values.

Each user has a *real* user ID and group ID and an *effective* user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. **exec** changes the effective user and group ID to the owner of the executed file if the file has the 'set-user-ID' or 'set-group-ID' modes. The real user ID is not affected.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg0*, *arg1*, ... *argn* address null-terminated strings. Conventionally *arg0* is the name of the file.

From C, two interfaces are available. **execl** is useful when a known file with known arguments is being called; the arguments to **execl** are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The **execv** version is useful when the number of arguments is unknown in advance; the arguments to **execv** are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

*argv* is directly usable in another **execv** because *argv*[*argc*] is 0.

*envp* is a pointer to an array of strings that constitute the *environment* of the process. Each string conventionally consists of a name, an '=', and a null-terminated value. The array of pointers is terminated by a null pointer. The shells **sh**(1) and **csh**(1) pass an environment entry for each global shell variable defined when the program is called. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by **execv** and **execl** to pass the environment to any subprograms executed by the current program. The **exec** routines use lower-level routines as follows to pass an environment explicitly:

```
execle(file, arg0, arg1, . . . , argn, 0, environ);
execve(file, argv, environ);
```

**execlp** and **execvp** are called with the same arguments as **execl** and **execv**, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

**FILES**

/bin/sh shell, invoked if command file found by **execlp** or **execvp**

**SEE ALSO**

fork(2), ioctl(2), signal(2), "VENIX Programming" in the *Programming Guide*.

## DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see **a.out**(4)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is $-1$. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

## BUGS

If **execvp** is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of **argv[0]** and **argv[** $-1$**]** will be modified before return.

## ASSEMBLER

(exec = 11.)

PDP-11:

```
sys exec; name; argv
```

8086:

```
BX = 11
AX = name
DX = argv
int 0Xf1
```

(exece = 59.)

PDP-11:

```
sys exece; name; argv; envp
```

8086:

```
BX = 59
AX = name
DX = argv
CX = envp
int 0Xf1
```

Plain **exec** is obsoleted by **exece**, but remains for historical reasons.

When the called file starts execution, the stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings, followed by a null pointer, followed by the pointers to the environment strings and then another null pointer. The strings themselves follow; a 0 word is left at the very top.

```
sp→   nargs
      arg0
      ...
      argn
      0
      env0
      ...
      envm
      0

arg0: <arg0\0>
      ...
env0: <env0\0>
      0
```

## NAME

exit — terminate process

## SYNOPSIS

**exit(status)**
**int status;**

**_exit(status)**
**int status;**

## DESCRIPTION

**exit** is the normal means of terminating a process. **exit** closes all the process' files and notifies the parent process if it is executing a **wait**(2). The low-order 8 bits of *status* are available to the parent process.

This call can never return.

The C function **exit** may cause cleanup actions before the final 'sys exit'. The function **_exit** circumvents all cleanup.

## SEE ALSO

wait(2)

## ASSEMBLER

(exit = 1.)

PDP-11:
R0 = status
sys exit

8086: BX = 1
AX = status
int 0Xf1

## NAME

fork — spawn new process

## SYNOPSIS

**fork( )**

## DESCRIPTION

**fork** is the only way new processes are created. The new process' core image is a copy of that of the caller of **fork**. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID is used by **wait**(2) and **kill**(2).

Files open before the **fork** are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

## SEE ALSO

wait(2), exec(2), kill(2)

## DIAGNOSTICS

Returns −1 and fails to create a process if: the user is not super-user, or the system's process table is full. Only the super-user can take the last process-table slot.

## ASSEMBLER

(fork = 2.)

PDP-11:

sys fork
(new process return)
(old process return, R0 = new process ID)
The return locations in the old and new process differ by one work. The C-bit is set in the old process if a new process could not be created.

8086: BX = 2
int 0Xf1
AX = 0 for child, child process ID for parent

## NAME

getpid — get process identification

## SYNOPSIS

**getpid( )**

## DESCRIPTION

**getpid** returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

## SEE ALSO

mktemp(3)

## ASSEMBLER

(getpid = 20.)

PDP-11:

```
sys getpid
R0 = process ID
```

8086:

```
BX = 20
int 0Xf1
AX = process ID
```

## NAME

getuid, getgid, geteuid, getegid — get user and group identity

## SYNOPSIS

**getuid( )**

**geteuid( )**

**getgid( )**

**getegid( )**

## DESCRIPTION

**getuid** returns the real user ID of the current process, **geteuid** the effective user ID. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the 'set user ID' mode, to find out who invoked them.

**getgid** returns the real group ID, **getegid** the effective group ID.

## SEE ALSO

setuid(2)

## ASSEMBLER

(getuid = 24.)

PDP-11:

```
sys getuid
R0 = real UID
R1 = effective UID
```

8086:

```
BX = 24
int 0Xf1
AX = real UID
DX = effective UID
```

(getgid = 47.)

PDP-11:

```
R0 = real GID
R1 = effective GID
```

8086: BX = 47
int 0Xf1
AX = real GID
DX = effective GID

**NAME**

indir — indirect system call

**ASSEMBLER**

(indir = 0.)

PDP-11:

```
sys indir; call
```

The system call at the location *call* is executed. Execution resumes after the **indir** call.

The main purpose of **indir** is to allow a program to store system calls and their arguments in the data segment. Since system calls are executed with a trap, their arguments must be placed directly after the **sys** instruction. In order to keep system call arguments in the data segment (and thus allow shared-text (pure) programs which must have totally separate text and data portions), the **indir** call is used to indirectly execute the system call in the data portion. The C interface for any system call with arguments uses this method.

If **indir** itself is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, **indir** returns error code EINVAL; see **intro**(2).

Because of **indir**'s special nature, it is executed at the assembler level only.

## NAME

ioctl, stty, gtty — control device

## SYNOPSIS

**#include <sgtty.h>**

**ioctl(fildes, request, argp)**
**struct sgttyb *argp;**

**stty(fildes, argp)**
**struct sgttyb *argp;**

**gtty(fildes, argp)**
**struct sgttyb *argp;**

## DESCRIPTION

**ioctl** performs a variety of functions on character special files (devices). The writeups on various devices in section 7, in the *Installation and System Manager's Guide*, discuss how **ioctl** applies to them.

For certain status setting and status inquiries about terminal devices, the functions **stty** and **gtty** are equivalent to

**ioctl(fildes, TIOCSETP, argp)**
**ioctl(fildes, TIOCGETP, argp)**

respectively; see **ttys**(4).

## SEE ALSO

stty(1), ttys(4), exec(2)

## DIAGNOSTICS

Zero is returned if the call was successful; −1 if the file descriptor does not refer to the kind of file for which it was intended.

## BUGS

Strictly speaking, since **ioctl** may be extended in different ways to devices with different properties, *argp* should have an open-ended declaration like

**union { struct sgttyb ...; ... } *argp;**

The important thing is that the size is fixed by 'struct sgttyb'.

**ASSEMBLER**

(ioctl = 54.)

PDP-11:

```
sys ioctl; fildes; request; argp
```

8086:

```
BX = 54
AX = fildes
DX = request
CX = argp
int 0Xf1
```

(stty = 31.)

PDP-11:

```
R0 = fildes
sys stty; argp
```

8086:

```
BX = 31
AX = fildes
DX = argp
int 0Xf1
```

(gtty = 32.)

PDP-11:

```
R0 = fildes
sys gtty; argp
```

8086:

```
BX = 32
AX = fildes
DX = argp
int 0Xf1
```

## NAME

kill — send signal to a process

## SYNOPSIS

**kill(pid, sig);**

## DESCRIPTION

**kill** sends the signal *sig* to the process specified by the process number *pid*. See **signal**(2) for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see **ttys**(4).

If the process number is −1, and the user is the super-user, the signal is broadcast universally except to processes 0 and 1, the scheduler and initialization processes. See **init**(8), section 8 in the *Installation and System Manager's Guide*.

Processes may send signals to themselves.

## SEE ALSO

signal(2), kill(1)

## DIAGNOSTICS

Zero is returned if the process is killed; −1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

## ASSEMBLER

(kill = 37.)

PDP-11:

```
R0 = process ID
sys kill; signal
```

8086:

```
BX = 37
AX = process ID
DX = signal
int 0Xf1
```

## NAME

libmon — library of system call routines for Pascal programs

## DESCRIPTION

The modules in this library comprise the VENIX system call interface for Pascal programs. System calls with their C-language interface are described in the other pages of this section of the manual. The Pascal interface is quite similar.

All calls are available with the following exceptions:

The system call **brk** is not available, because the memory allocation for Pascal programs is quite different.

The system call **signal** is replaced by **sigtrp**, with the following calling sequence:

```
function sigtrp(signo,trapno:integer):integer;
```

One of the reasons is that the action values of **signal**, odd for 'ignore' and zero for 'get back to default', interfere with the Pascal procedure identification. Procedures in Pascal are numbered consecutively from zero up. The first argument of **sigtrp** is the signal number *signo* as for **signal**. The second argument is an integer *trapno*, indicating the action to be performed when the signal is issued:

−2 Reset the action for signal *signo* to the default.

−3 Ignore signal *signo*.

0−255 Perform an EM-1 instruction TRP with error code *trapno*, whenever the signal *signo* is issued. Note that the error codes 0−127 are reserved for EM-1 machine errors and language run-time system errors.

The routine **sigtrp** returns the previous *trapno* or −1 if an erroneous signal number is specified. Only the signal numbers 1, 2, 3, 13, 14, 15 and 16 may be used as argument for *sigtrp*.

## FILES

| | |
|---|---|
| /usr/lib/libmon.a | the version for compiled programs |
| /usr/lib/em1_mon.a | the version for interpreted programs |

**SEE ALSO**

em1(1), pc(1), libpc(3)

**DIAGNOSTICS**

All routines put the VENIX error code in the global variable **errno**. **errno** is not cleared by successful system calls, so it always gives the error of the last failed call. One exception: **ptrace**(2) clears **errno** when successful.

**AUTHOR**

Johan Stevenson, Vrije Universiteit

**BUGS**

There should be additional routines giving a fatal error when they fail. In C you are allowed to call a function without testing its result. In Pascal you have stronger typė checking. In these circumstances it would be pleasant to have routines which print a nice message and stop execution for unexpected errors.

## NAME

link — link to a file

## SYNOPSIS

**link(name1, name2)**
**char *name1, *name2;**

## DESCRIPTION

A link to *name1* is created; the link has the name *name2*. Either name may be an arbitrary pathname.

## SEE ALSO

ln(1), unlink(2)

## DIAGNOSTICS

Zero is returned when a link is made; −1 is returned when *name1* cannot be found; when *name2* already exists; when the directory of *name2* cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when a file has too many links.

## ASSEMBLER

(link = 9.)

PDP-11:

```
sys link; name1; name2
```

8086:

```
BX = 9
AX = name1
DX = name2
int 0Xf1
```

## NAME

lock — lock a process in primary memory

## SYNOPSIS

**lock(flag)**

## DESCRIPTION

If the *flag* argument is non-zero, the process executing this call will not be swapped out of memory except if it is required to grow. If the argument is zero, the process is un**lock**ed. This call may only be executed by the super-user or if the caller's group ID is zero.

Processes are removed from memory when they exit.

## BUGS

**lock**ed processes interfere with the compaction of primary memory and can cause deadlock.

## DIAGNOSTICS

Zero is returned if the call is successful; −1 if not.

## ASSEMBLER

(lock = 53.)

PDP-11:

```
sys lock; flag
```

8086:

```
BX = 53
AX = flag
int 0Xf1
```

## NAME

lseek, tell — move read/write pointer

## SYNOPSIS

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

**long tell(fildes)**

## DESCRIPTION

The file descriptor *fildes* refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the end of the file plus *offset*.

The returned value is the resulting pointer location.

The obsolete function **tell(***fildes***)** is identical to **lseek(***fildes***, 0L, 1).**

Seeking far beyond the end of a file, then writing, creates a gap or 'hole', which occupies no physical space and reads as zeros.

## SEE ALSO

open(2), creat(2), fseek(3)

## DIAGNOSTICS

−1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

## BUGS

**lseek** is a no-op on character special files.

## ASSEMBLER

(lseek = 19.)

PDP-11:

R0 = fildes
sys lseek; offset1; offset2; whence

*offset1* and *offset2* are the high and low words of *offset*; R0 and R1 contain the pointer upon return.

8086:

BX = 19
AX = file descriptor
DX = offset 1
CX = offset 2
SI = whence
int 0Xf1

*offset1* and *offset2* are the high and low words of *offset*; AX and DX contain the pointer upon return.

## NAME

mknod — make a directory or a special file

## SYNOPSIS

**mknod(name, mode, addr)**
**char *name;**

## DESCRIPTION

**mknod** creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory and special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process' mode mask; see **umask**(2)). The first block pointer of the i-node is initialized from *addr*. For ordinary files and directories *addr* is normally zero. In the case of a special file, *addr* specifies which special file ((majnum < < 8) | minnum).

**mknod** may be invoked only by the super-user.

## SEE ALSO

mkdir(1), mknod(1), filsys(4)

## DIAGNOSTICS

Zero is returned if the file has been made; −1 if the file already exists or if the user is not the super-user.

## ASSEMBLER

(mknod = 14.)

PDP-11:

sys mknod; name; mode; addr

8086: BX = 14
AX = name
DX = mode
CX = addr
int 0Xf1

## NAME

mount, umount — mount or remove file system

## SYNOPSIS

**mount(special, name, rwflag)**
**char *special, *name;**

**umount(special)**
**char *special;**

## DESCRIPTION

**mount** announces to the system that a removable file system has been mounted on the block-structured special file *special*; from now on, references to file *name* will refer to the root file on the newly mounted file system. *special* and *name* are pointers to null-terminated strings containing the appropriate pathnames.

*name* must exist already. *name* must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

**umount** announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

## SEE ALSO

mount(1)

## DIAGNOSTICS

**mount** returns 0 if the action occurred; −1 if *special* is inaccessible or not an appropriate file; if *name* does not exist; if *special* is already mounted; if *name* is in use; or if there are already too many file systems mounted.

**umount** returns 0 if the action occurred; −1 if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

**ASSEMBLER**

(mount = 21.)

PDP-11:

```
sys mount; special; name; rwflag
```

8086:

```
BX = 21
AX = special
DX = name
CX = rwflag
int 0Xf1
```

(umount = 22.)

PDP-11:

```
sys umount; special
```

8086:

```
BX = 22
AX = special
int 0Xf1
```

## NAME

nice — set program priority

## SYNOPSIS

**nice(incr)**

## DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the super-user or users with group ID's of zero. The priority is limited to the range −20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by **fork**(2). For a privileged process to return to normal priority from an unknown state, **nice** should be called successively with arguments −40 (goes to priority −20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

If the increment is −100 or less and the user is a super-user, then the process gains "real-time" ("pre-emptive") priority. This means that all CPU time and disk queuing resources are made exclusively available to this process. 'Normal' processes will not run unless the 'real-time' process **sleeps**(3) for some interval, does synchronous I/O (e.g. terminal input or a lot of disk I/O), **exits**, or turns off 'real-time' by re-calling **nice** with a positive increment. 'Real-time' processes can be swapped, thus the process probably should also **lock**(2) itself into memory. Simultaneous 'real-time' processes are scheduled round-robin on a one clock-tick interval. 'Real-time' characteristics are not inherited by children.

## SEE ALSO

nice(1)

## NOTES

'Real-time' priorities are not portable to standard UNIX.

**ASSEMBLER**

(nice = 34.)

PDP-11:

```
R0 = priority
sys nice
```

8086:

```
BX = 34
AX = priority
int 0Xf1
```

## NAME

open — open for reading or writing

## SYNOPSIS

**open(name, mode)**
**char *name;**

## DESCRIPTION

**open** opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

**open** returns a file descriptor which must be used in subsequent calls for other input-output functions on the file.

The file pointer is positioned at the beginning of the file (byte 0).

## SEE ALSO

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

## DIAGNOSTICS

The value −1 is returned if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open.

## ASSEMBLER

(open = 5.)

PDP-11:

```
sys open; name; mode
R0 = file descriptor
```

8086:

```
BX = 5
AX = name
DX = mode
int 0Xf1
AX = file descriptor
```

## NAME

pause — stop until signal

## SYNOPSIS

**pause( )**

## DESCRIPTION

**pause** never returns normally. It is used to give up control while waiting for a signal from **kill**(2) or **alarm**(2).

## SEE ALSO

kill(1), kill(2), alarm(2), signal(2), setjmp(3)

## ASSEMBLER

(pause = 29.)

PDP-11:

```
sys pause
```

8086:

```
BX = 29
int 0Xf1
```

## NAME

phys — allow a process to access physical addresses

## SYNOPSIS

PDP-11:

**phys(segreg, size, physadr)**

8086: **phys(0, 0, physadr)**

## DESCRIPTION

**phys** allows a process to access physical memory, normally not in the process' address space. This call is obviously machine dependent and very dangerous. Its arguments and actions differ somewhat between PDP-11 and 8086 processors:

PDP-11:

The argument *segreg* specifies a process virtual (data-space) address range of 8K bytes starting at virtual address *segreg*×8K bytes. This address range is mapped into physical address *physadr*×64 bytes. Only the first *size*×64 bytes of this mapping is addressable. If *size* is zero, any previous mapping of this virtual address range is nullified. For example, the call

```
phys(6, 1, 0177775);
```

will map virtual addresses 0140000 – 0140077 into physical addresses 017777500 – 017777577. In particular, virtual address 0140060 is the PDP-11 console located at physical address 017777560.

8086: The user's extra segment is mapped into physical address *physadr*×512 bytes. If *physadr* is −1, any previous mapping of this virtual address range is nullified. For example, the call

```
phys(0, 0, 0XB800/(512/16));
```

will map extra segment addresses 0 to 0XFFFF into physical addresses B8000 – C7FFFF.

After mapping via **phys**, the extra segment may be read and written with the calls described in **getes**(3).

This call may only be executed by the super-user or if the caller's group ID is zero.

**SEE ALSO**

getes(3), sdata(2)

**DIAGNOSTICS**

The function value zero is returned if the physical mapping is in effect. The value −1 is returned if not super-user, or super group, or an **sdata** call is in effect.

**ASSEMBLER**

(phys = 52.)

PDP-11:

```
sys phys; segreg; size; physadr
```

8086:

```
BX = 52
AX = ?
DX = ?
CX = physadr
int 0Xf1
```

## NAME

pipe — create an interprocess channel

## SYNOPSIS

**pipe(fildes)**
**int fildes[2];**

## DESCRIPTION

The **pipe** system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes*[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes*[0] will pick up the data. Writes with a count of 4096 bytes or less are atomic; no other process can intersperse data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent **fork**(2) calls) will pass data through the pipe with **read**(2) and **write**(2) calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

## SEE ALSO

sh(1), read(2), write(2), fork(2), popen(3)

## DIAGNOSTICS

The function value zero is returned if the pipe was created; −1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

## BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

**ASSEMBLER**

(pipe = 42.)

PDP-11:

sys pipe
R0 = read file descriptor
R1 = write file descriptor

8086: BX = 42
int 0Xf1
AX = read file descriptor
BX = write file descriptor

## 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 (frequency HZ is machine dependent); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777(8) gives a 1 – 1 mapping of pc values to words in *buff*; 077777(8) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an **exec**(2) is executed, but remains on in child and parent both after a **fork**(2). Profiling may be turned off if an update in *buff* would cause a memory fault.

## SEE ALSO

monitor(3), prof(1), exec(2), fork(2)

## ASSEMBLER

(profil = 44.)

PDP-11:

```
sys profil; buff; bufsiz; offset; scale
```

8086:

```
BX = 44
AX = buff
DX = buffsiz
CX = offset
SI = scale
int 0Xf1
```

## NAME

ptrace — process trace

## SYNOPSIS

#include <signal.h>

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

## DESCRIPTION

**ptrace** provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt.' See **signal**(2) for the list. Then the traced process enters a stopped state and its parent is notified via **wait**(2). When the child is in the stopped state, its core image can be examined and modified using **ptrace**. If desired, another **ptrace** request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

**0** This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

**1,2** The word in the child process' address space at *addr* is returned. If I and D space are separated, request 1 indicates I space, 2 D space. On the PDP-11, *addr* must be even. The child must be stopped. The input *data* is ignored.

**3** The word of the system's per-process data area corresponding to *addr* is returned. *addr* must be even and less than 512 (PDP-11) or 1024 (8086). This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

**4,5** The given *data* is written at the word in the process' address space corresponding to *addr*, which must be even. No useful value is

returned. If I and D space are separated, request 4 indicates I space, 5 D space. On the PDP-11 only, attempts to write in a pure procedure fail if another proess is executing the same file.

**6** The process' system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

**7** The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process' image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

**8** The traced process terminates.

**9** Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. On the PDP-11, the T-bit is used and just one instruction is executed. This is part of the mechanism for implementing breakpoints.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The **wait** call is used to determine when a process stops; in such a case the 'termination' status returned by **wait** has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, **ptrace** inhibits the set-user-id facility on subsequent **exec**(2) calls. If a traced process calls **exec**, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

## SEE ALSO

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

## DIAGNOSTICS

The value $-1$ is returned if *request* is invalid, *pid* is not a traceable process, *addr* is out of bounds, or *data* specifies an illegal signal number.

## BUGS

The error indication, $-1$, is a legitimate function value; **errno**, see **intro**(2), can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

**ASSEMBLER**

(ptrace = 26.)

PDP-11:

```
R0 = data
sys ptrace; pid; addr; request
R0 = value
```

8086:

```
BX = 26
AX = data
DX = pid
CX = addr
SI = request
int 0Xf1
AX = value
```

## NAME

read — read from file

## SYNOPSIS

**read(fildes, buffer, nbytes)**
**char *buffer;**

## DESCRIPTION

*fildes*, a file descriptor, is an integer returned by a successful **open**, **creat**, **dup**, or **pipe**(2) call. *buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a terminal at most one line will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached immediately, with no bytes read.

## SEE ALSO

open(2), creat(2), dup(2), pipe(2)

## DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is −1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes*, file descriptor not that of an input file.

## ASSEMBLER

(read = 3.)

PDP-11:
R0 = fildes
sys read; buffer; nbytes
R0 = byte count

8086: BX = 3
AX = fildes
DX = buffer
CX = nbytes
int 0Xf1
AX = byte count

## NAME

sdata — manipulate a shared data segment

## SYNOPSIS

PDP-11:

```
sdata(arg, reg, offset)
char *arg;
```

8086:

```
sdata(arg, 0, offset)
char *arg;
```

## DESCRIPTION

**sdata** manipulates a shared data segment. On the PDP-11, the segment is placed in the 8kb user data segment indicated by argument *reg*. On the 8086, the extra memory segment is used.

The operation is given by *arg*:

*filename*

If *arg* is a file name (null terminated string), then a "named" segment is opened; that file is windowed into the shared segment. If this is the first process to call up the file, then the file is first read into memory; if the file has already been **sdata**'d by another process, then it is hooked up without further I/O. The window will be placed into the file at an initial offset given by *offset*×64 bytes (PDP-11) or *offset*×512 bytes (8086).

The size of the shared data segment is given by the length of the file, rounded up to the next 64 (PDP-11) or 512 (8086) byte boundary.

**(char *) 0**

The window offset into the previously hooked-to segment is changed to *offset*×64 (PDP-11) or *offset*×512 (8086) bytes. This allows the user to move his window to any location in the segment.

**(char *) 1**

An "unnamed" segment is opened. This segment is not associated with any disk file, and can not be shared by multiple processes; it merely allows an individual process to hook into an extra memory area. *offset* is the length of the segment, in 64 (PDP-11) or 512 (8086) byte units.

**(char *) 2**
(reserved for future use)

**(char *) 3**
The previously opened segment (named or unnamed) is closed, and memory is unmapped for the calling process. Closing a named segment does not affect any other processes hooked to the same segment. When a named segment is no longer held open by any processes, it is dropped from memory unless the associated file has the 'sticky bit' set (mode 01000 – see **chmod**(2)). In this case the segment remains intact forever.

The user is responsible for making sure that the memory mapped for the shared data segment is not otherwise used by his program. The **sdata** call returns an error if it is. See **phys**(2).

Only one shared data segment per process can be hooked to at a time. If several processes may be simultaneously writing to the same area at once, you will probably wish to use semaphores (see **semset**(2)) to prevent conflicts.

**PDP-11 NOTES**
After hooking to a shared data segment, PDP-11 programs may access it as part of their normal user memory. On the PDP-11, the top 8kb of virtual memory (register 7) is always reserved for the program stack, and should never be used; however, register 6 will be available unless the program is very large, or unless it has been previously allocated by **phys**(2) (see also note below concerning **phys**). The **size**(1) command can be used to determine the amount of space used by your program.

All free memory pages can be mapped into the same shared file (presumably with different offsets) by opening the shared file, and then setting *arg* to zero and giving different values of *reg* on subsequent **sdata** calls.

The mapping of phyical memory (with **phys**(2)) to another register may be lost when a shared segment is opened; to be safe, if **phys** calls are needed they should be done after opening the shared segment.

**8086 NOTES**
On the 8086, the **getes** and **putes** calls may be used to read and write to the shared segment once it is hooked to. See **getes**(3).

**SEE ALSO**

phys(2), semset(2), getes(3)

**DIAGNOSTICS**

On error −1 is returned.

Shared data segments is a feature of VENIX which is not portable to standard UNIX.

**ASSEMBLER**

(sdata = 49.)

PDP-11:

```
sys sdata; file; reg; offset
```

8086:

```
BX = 49
AX = file
DX = ?
CX = offset
int 0Xf1
```

## NAME

semset, semclear, semtest, semtset — manipulate local/global binary semaphores

## SYNOPSIS

**semset(sem, pri)**
**semclear(sem)**
**semtest(sem)**
**semtset(sem, pri)**

## DESCRIPTION

Semaphores allow cooperating processes to "lock out" each other during the execution of "critical code" regions, such as during updates to shared data segments or any common data base.

**semset** sets the semaphore *sem* if it was clear and returns to the caller; otherwise it queues the calling process at priority *pri* with all other processes waiting on *sem* and goes to sleep. The values of *pri* may range between 0, the highest priority, and 15, the lowest priority.

**semclear** clears the semaphore previously set and wakes up the highest priority process waiting on *sem*.

**semtest** tests the semaphore and returns a zero if clear, a one if set. If clear, this does not guarantee that a subsequent **semset** will not have to wait, since another process can do a **semset** in the intervening time.

**semtset** tests the semaphore and returns a 1 if set. If clear, then the semaphore is set and a 0 is returned.

Semaphores can range in value between −16 and 15. The negative values (−16 to −1) are global and the same for processes on the system, while the positive values (0 to 15) are local and shared by all processes in the same process group.

When a process forks, only the parent maintains the semaphore. Semaphores are maintained beyond the life of any program using them; they are not cleared when a program exits.

## DIAGNOSTICS

A semaphore out of range or attempted setting if already set by the caller is considered an error and a −1 is returned.

**BUGS**

If a process is swapped out while waiting on a semaphore, its priority is ineffective for waking up.

**NOTES**

Semaphores are not portable to standard UNIX.

**ASSEMBLER**

(sema = 45.)

PDP-11:

R0 = 0 (set), 1 (clear), 2 (test), or 3 (test & set)
sys sema; semaphore; priority

8086: BX = 45
AX = 0 (set), 1 (clear), 2 (test), or 3 (test & set)
DX = semaphore
CX = priority
int 0Xf1

## NAME

setuid, setgid — set user and group ID

## SYNOPSIS

**setuid(uid)**

**setgid(gid)**

## DESCRIPTION

The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user or if the argument is the real ID.

## SEE ALSO

getuid(2)

## DIAGNOSTICS

Zero is returned if the user (group) ID is set; −1 is returned otherwise.

## ASSEMBLER

(setuid = 23.)

PDP-11:

```
R0 = uid
sys setuid
```

8086:

```
BX = 23
AX = uid
int 0Xf1
```

(setgid = 46.)

PDP-11:

```
R0 = gid
sys setgid
```

8086:

```
BX = 46
AX = gid
int 0Xf1
```

## NAME

signal — catch or ignore signals

## SYNOPSIS

**#include <signal.h>**

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

## DESCRIPTION

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

| | | |
|---|---|---|
| SIGHUP | 1 | hangup |
| SIGINT | 2 | interrupt |
| SIGQUIT | 3* | quit |
| SIGILL | 4* | illegal instruction (not reset when caught) |
| SIGTRAP | 5* | trace trap (not reset when caught) |
| SIGIOT | 6* | IOT instruction or asynchronous i/o error |
| SIGEMT | 7* | EMT instruction |
| SIGFPE | 8* | floating point exception |
| SIGKILL | 9 | kill (cannot be caught or ignored) |
| SIGBUS | 10* | bus error |
| SIGSEGV | 11* | segmentation violation |
| SIGSYS | 12* | bad argument to system call |
| SIGPIPE | 13 | write on a pipe or link with no one to read it |
| SIGALRM | 14 | alarm clock |
| SIGTERM | 15 | software termination signal |
| SIGAIO | 16 | asynchronous i/o completed |

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

*sig* must be one of the signal numbers given above. *func* is either a pointer to a function or one of the special values SIG_DFL or SIG_IGN.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination, sometimes with a core image. If *func* is SIG_IGN the signal is ignored. Otherwise when the signal occurs *func* will be

called with the signal number as argument. A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another **signal** call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a **read** or **write**(2) on a slow device (like a terminal; but not a file); and during **pause** or **wait**(2). When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of **signal** is the previous (or initial) value of *func* for the particular signal.

After a **fork**(2) the child inherits all signals. **exec**(2) resets all caught signals to default action.

**SEE ALSO**

kill(1), kill(2), ptrace(2), setjmp(3)

**DIAGNOSTICS**

The value (int) – 1 is returned if the given signal is out of range.

**BUGS**

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its *func* argument are problematical.

**ASSEMBLER**

(signal = 48.)

PDP-11:

```
sys signal; sig; label
R0 = old label
```

```
8086:   BX = 48
        AX = sig
        DX = label
        int 0Xf1
        AX = old label
```

If *label* is 0, default action is reinstated. If *label* is 1, the signal is ignored. Any other *label* specifies an address in the process where an interrupt is simulated. An IRET instruction will return from the interrupt.

## NAME

stat, fstat — get file status

## SYNOPSIS

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

stat(name, buf)
char *name;
struct stat *buf;

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

## DESCRIPTION

**stat** obtains detailed information about a named file. **fstat** obtains the same information about an open file known by the file descriptor from a successful **open**, **creat**, **dup**, or **pipe**(2) call.

*name* points to a null-terminated string naming a file; *buf* is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable. The layout of the structure pointed to by *buf* as defined in <**sys/stat.h**> is given below. **st_mode** is encoded according to the "#define" statements.

```
struct    stat
{
        dev_t     st_dev;
        ino_t     st_ino;
        unsigned short    st_mode;
        short     st_nlink;
        short     st_uid;
        short     st_gid;
        dev_t     st_rdev;
        off_t     st_size;
        time_t    st_atime;
        time_t    st_mtime;
        time_t    st_ctime;
};
```

The meaning of each element is:

**st_dev** major/minor number of device this file is on

| | |
|---|---|
| **st_ino** | inode number of this file |
| **st_mode** | file mode (see encoding below) |
| **st_uid** | owner ID number |
| **st_gid** | group ID number |
| **st_rdev** | if this is a special file (device), major/minor number of device it points to |
| **st_size** | length in bytes |
| **st_atime** | last accessed time (for reasons of efficiency, this is not set when a directory is searched, although this would be more logical) |
| **st_mtime** | last modified time |
| **st_ctime** | currently the same as **st_mtime** |

The bit encoding of **st_mode** is

```
#define  S_IFMT    0160000    /* type of file */
#define  S_IFDIR   0140000    /* directory */
#define  S_IFCHR   0120000    /* character special */
#define  S_IFBLK   0160000    /* block special */
#define  S_IFREG   0100000    /* regular */
#define  S_ILRG    0010000    /* large file */
#define  S_ISUID   0004000    /* set user id on execution */
#define  S_ISGID   0002000    /* set group id on execution */
#define  S_ISVTX   0001000    /* save shared even after use */
#define  S_IREAD   0000400    /* read permission, owner */
#define  S_IWRITE  0000200    /* write permission, owner */
#define  S_IEXEC   0000100    /* execute/search permission */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see **chmod**(2)).

The defined types, **ino_t**, **off_t**, **time_t**, name various width integer values; **dev_t** encodes major and minor device numbers; their exact definitions are in the include file <sys/types.h> (see **types**(5)).

When *fildes* is associated with a pipe, **fstat** reports an ordinary file with restricted permissions. The size is the number of bytes queued in the pipe.

Note that the stat buffer format differs from the disk inode format.

**SEE ALSO**

ls(1), filsys(4)

**DIAGNOSTICS**

Zero is returned if a status is available; −1 if the file cannot be found.

**ASSEMBLER**

(stat = 18.)

PDP-11:

```
sys stat; name; buf
```

8086:

```
BX = 18
AX = name
DX = buf
int 0Xf1
```

(fstat = 28.)

PDP-11:

```
R0 = file descriptor
sys fstat; buf
```

8086:

```
BX = 28
AX = buf
int 0Xf1
```

## NAME

stime — set time

## SYNOPSIS

**stime(tp)**
**long *tp;**

## DESCRIPTION

**stime** sets the system's idea of the time and date. Time, pointed to by *tp*, is measured in seconds from 00:00:00 GMT Jan 1, 1970. Only the super-user may use this call.

## SEE ALSO

date(1), time(2), ctime(3)

## DIAGNOSTICS

Zero is returned if the time was set; −1 if user is not the super-user.

## ASSEMBLER

(stime = 25.)

PDP-11:

```
R0, R1 = time
sys stime
```

8086:

```
BX = 25
AX = time1
DX = time2
int 0Xf1
```

## NAME

suspend — suspend/resume a process

## SYNOPSIS

**suspend(pid, flag)**

## DESCRIPTION

**suspend** suspends a process specified by *pid* if *flag* is non-zero, or resumes a process specified by *pid* if *flag* is zero. The sending and receiving process must have the same effective user ID; otherwise this call is restricted to the super-user.

While a process is suspended, it can only be terminated by a kill signal. One of any other signal sent the process will be caught and acted upon when the process is resumed; more than one of a particular signal sent a suspended process will be ignored.

## SEE ALSO

suspend(1), kill(2)

## DIAGNOSTICS

A −1 is returned if the process does not exist or if the process does not have the same effective user ID and the user is not the super-user.

## NOTES

Process suspension is a feature of VENIX which is not portable to standard UNIX.

## ASSEMBLER

(suspend = 50.)

PDP-11:

```
R0 = pid
sys suspend; flag
```

8086:

```
BX = 50
AX = pid
DX = flag
int 0Xf1
```

## NAME

sync — update super-block

## SYNOPSIS

**sync( )**

## DESCRIPTION

**sync** causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example **fsck**(1), **df**(1), etc. A **sync** is done automatically when a programs exits.

## BUGS

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

## ASSEMBLER

(sync = 36.)

PDP-11:

```
sys sync
```

8086:

```
BX = 36
int 0Xf1
```

## NAME

time, ftime — get date and time

## SYNOPSIS

**long time(0)**

**long time(tloc)**
**long *tloc;**

**#include <sys/types.h>**
**#include <sys/timeb.h>**
**ftime(tp)**
**struct timeb *tp;**

## DESCRIPTION

**time** returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

The **ftime** entry fills in a structure pointed to by its argument, as defined by **<sys/timeb.h>**:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
        time_t time;
        unsigned short millitm;
        short timezone;
        short dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that daylight saving time applies locally during the appropriate part of the year.

## SEE ALSO

date(1), stime(2), ctime(3)

**ASSEMBLER**

(ftime = 35.)

PDP-11:

```
sys ftime; bufptr
```

BX = 35

```
AX = bufptr
int 0Xf1
```

(time = 13.; obsolete call)

PDP-11:

```
sys time
R0, R1 = time since 1970
```

8086:

```
BX = 13
int 0Xf1
AX, DX = time since 1970
```

## NAME

times — get process times

## SYNOPSIS

**times(buffer)**
**struct tbuffer *buffer;**

## DESCRIPTION

**times** returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/Hz seconds, where Hz = 60.

After the call, the buffer will appear as follows:

```
struct tbuffer {
        long    proc_user_time;
        long    proc_system_time;
        long    child_user_time;
        long    child_system_time;
};
```

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

## SEE ALSO

time(1), time(2)

## ASSEMBLER

(times = 43.)

PDP-11:
        sys times; buffer

8086:   BX = 43
        AX = buffer
        int 0Xf1

## NAME

umask — set file creation mode mask

## SYNOPSIS

**umask(complmode)**

## DESCRIPTION

**umask** sets a mask used whenever a file is created by **creat**(2) or **mknod**(2): the actual mode (see **chmod**(2)) of the newly-created file is the logical **and** of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 0 (no restrictions). The mask is inherited by child processes.

## SEE ALSO

creat(2), mknod(2), chmod(2)

## ASSEMBLER

(umask = 60.)

PDP-11:

```
sys umask; complmode
```

8086:

```
BX = 60
AX = complmode
int 0Xf1
```

## NAME

unlink — remove directory entry

## SYNOPSIS

**unlink(name)**
**char *name;**

## DESCRIPTION

*name* points to a null-terminated string. **unlink** removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

## SEE ALSO

rm(1), link(2)

## DIAGNOSTICS

Zero is normally returned; −1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

## ASSEMBLER

(unlink = 10.)

PDP-11:

```
sys unlink; name
```

8086:

```
BX = 10
AX = name
int 0Xf1
```

## NAME

utime — set file times

## SYNOPSIS

```
#include <sys/types.h>
utime(file, timep)
char *file;
time_t timep[2];
```

## DESCRIPTION

The **utime** call uses the 'accessed' and 'updated' times in that order from the *timep* vector to set the corresponding recorded times for *file*.

The caller must be the owner of the file or the super-user.

Actually, the 'accessed' time is always set to the current time.

## SEE ALSO

stat(2)

## ASSEMBLER

(utime = 30.)

PDP-11:

```
sys utime; file; timep
```

8086:

```
BX = 30
AX = file
DX = timep
int 0Xf1
```

## NAME

wait — wait for process to terminate

## SYNOPSIS

**wait(status)**
**int *status;**

**wait(0)**

## DESCRIPTION

**wait** causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last **wait**, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of −1 returned). The normal return yields the process ID of the terminated child. In the case of several children, several **wait** calls are needed to learn of all the deaths.

If (int)*status* is nonzero, the high byte of the word pointed to receives the low byte of the argument of **exit**(2) when the child terminated. The low byte receives the termination status of the process. See **signal**(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See **ptrace**(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

## SEE ALSO

exit(2), fork(2), signal(2)

## DIAGNOSTICS

Returns −1 if there are no children not previously waited for.

## ASSEMBLER

(wait = 7.)

PDP-11:

sys wait
R0 = process ID
R1 = status

8086: BX = 7
int 0Xf1
AX = process ID
DX = status

The high byte of the status is the low byte of AX in the child at termination.

## NAME

write — write on a file

## SYNOPSIS

**write(fildes, buffer, nbytes)**
**char *buffer;**

## DESCRIPTION

The file descriptor *fildes* is an integer returned from a successful **open**, **creat**, **dup**, or **pipe**(2) call.

*buffer* is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

## SEE ALSO

creat(2), dup(2), open(2), pipe(2)

## DIAGNOSTICS

Returns −1 on error: bad descriptor, buffer address, or count; physical I/O errors.

## ASSEMBLER

(write = 4.)

PDP-11:

```
R0 = fildes
sys write; buffer; nbytes
R0 = byte count
```

8086:

```
BX = 4
AX = fildes
DX = buffer
CX = nbytes
int 0Xf1
AX = byte count
```

3

## NAME

intro — introduction to library functions

## SYNOPSIS

**#include <stdio.h>**

**#include <math.h>**

## DESCRIPTION

This section describes functions that may be found in various libraries, other than those functions that directly invoke VENIX system primitives (i.e. system calls), which are described in section 2.

These functions are directly callable by C programs; use of many of them is discussed in the chapter "VENIX Programming" in the *Programming Guide*. These functions are also callable by Fortran programs (available on PRO/VENIX and VENIX/11 only), so long as care is taken to match the C argument sequence appropriately. This is described in the Fortran 77 document within the same guide. Routines in pages marked (3P) are callable exclusively by Pascal programs (available on PRO/VENIX and VENIX/11 only).

One page heading in this section may cover a number of related functions. The cross-reference in the following pages can be used to locate the page a particular function is on.

Functions are divided into various libraries distinguished by the section number at the top of the page:

(3) These functions, together with those of section 2 and those marked (3S), constitute library **libc**, which is automatically loaded by the C compiler **cc**(1). Some are callable from Fortran as well. The link editor **ld**(1) searches this library under the '−lc' option (this is automatically done by the C compiler and Fortran compilers.) Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3G) These functions are part of the graphics libraries, a set of plotting routines callable by C. Several versions of the libraries exist; see **plot**(3G).

(3M) These functions constitute the math library, **libm**. The link editor searches this library under the '−lm' option (this should be given

at the end of the **cc** command lines). Declarations for these functions may be obtained from the include file <**math.h**>.

(3P) These functions are part of VU-Pascal (PRO/VENIX and RAINBOW/VENIX only). They can be called exclusively by Pascal programs.

(3S) These functions constitute the 'standard I/O package'; see **stdio**(3). These functions are in the library **libc** already mentioned. Declarations for these functions may be obtained from the include file <**stdio.h**>.

(3X) Various specialized libraries have not been given distinctive captions. The files in which these libraries are found are named on the appropriate pages. The flag "−lxxx" should be used at the end of the compiler command line when using library "xxx".

The "SYNOPSIS" sections indicate the types of arguments that the given function expects, and the value it returns. For example, **atof** converts character strings into double precision numbers. It is listed

```
double atof(nptr)
char *nptr;
```

This means that **atof()** returns a value of type **double**; the argument **nptr** is a pointer to **char**, (i.e., a character string). Since **atof** returns a non-integer value, the function itself should be declared prior to use as

```
double atof();
```

The notation

```
#include <header.h>
```

at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling the given function. These headers contain definitions for constants and macro functions, and type declarations for subroutines.

## FILES

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

## SEE ALSO

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

## DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable **errno** (see **intro**(2)) is set to the value EDOM or ERANGE. The values of EDOM and ERANGE are defined in the include file <**errno.h**>.

## NAME

abort — generate IOT fault

## DESCRIPTION

**abort** executes the int 0Xf3 (8086) or IOT (PDP-11) instruction. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

## SEE ALSO

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

## DIAGNOSTICS

Usually 'IOT trap – core dumped' from the shell.

## NAME

abs — integer absolute value

## SYNOPSIS

**abs(i)**

## DESCRIPTION

**abs** returns the absolute value of its integer operand.

## SEE ALSO

floor(3) for **fabs**

## BUGS

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

## NAME

assert — program verification

## SYNOPSIS

**#include <assert.h>**

**assert (expression)**

## DESCRIPTION

**assert** is a macro that indicates *expression* is expected to be true at this point in the program. It causes an **exit**(2) with a diagnostic comment on the standard output when *expression* is false (=0). Compiling with the **cc**(1) option **−DNDEBUG** effectively deletes **assert** from the program.

## DIAGNOSTICS

'Assertion failed: file *f* line *n*.' *f* is the source file and *n* the source line number of the **assert** statement.

## NAME

atof, atoi, atol — convert ASCII to numbers

## SYNOPSIS

**double atof(nptr)**
**char *nptr;**

**atoi(nptr)**
**char *nptr;**

**long atol(nptr)**
**char *nptr;**

## DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

**atof** recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

**Atoi** and **atol** recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

## SEE ALSO

scanf(3)

## BUGS

There are no provisions for overflow.

## NAME

crypt, encrypt — a one way hashing encryption algorithm

## SYNOPSIS

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

**encrypt(block)**
**char *block;**

## DESCRIPTION

**crypt** is the password encryption routine. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

*key* is a user's typed password. *salt* is a two-character string chosen from the set [a–zA–Z0–9./]. The *salt* string is used to perturb the hashing 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. The first two characters are the salt itself.

There is a character array of length 64 containing only the numerical value 0 and 1. When this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine by **crypt**.

The **encrypt** entry provides (rather primitive) access to the actual hashing algorithm. The argument to the **encrypt** entry is a character array of length 64 containing only the characters with numerical value of 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by **crypt**.

## SEE ALSO

passwd(1), passwd(4), login(1), getpass(3)

## BUGS

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

## NAME

ctime, localtime, gmtime, asctime, timezone — convert date and time to ASCII

## SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

## DESCRIPTION

**ctime** converts a time pointed to by *clock* such as returned by **time**(2) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

Sun Sep 16 01:03:52 1973\n\0

**localtime** and **gmtime** return pointers to structures containing the broken-down time. **localtime** corrects for the time zone and possible daylight savings time; **gmtime** converts directly to GMT, which is the time VENIX 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 <**time.h**> is:

```
struct tm {
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1 – 31), month of year (0 – 11), day of week (Sunday = 0), year – 1900, day of year (0 – 365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the standard U.S.A. daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

**timezone** returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the daylight saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g. in Afghanistan **timezone(–(60*4+30), 0)** is appropriate because it is 4:30 ahead of GMT and the string **GMT+4:30** is produced.

## SEE ALSO

time(2)

## BUGS

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

## NAME

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii — character classification

## SYNOPSIS

**#include <ctype.h>**

**isalpha(c)**

. . .

## DESCRIPTION

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. **isascii** is defined on all integer values; the rest are defined only where **isascii** is true and on the single non-ASCII value EOF (see **stdio**(3)).

| | |
|---|---|
| **isalpha** | *c* is a letter |
| **isupper** | *c* is an upper case letter |
| **islower** | *c* is a lower case letter |
| **isdigit** | *c* is a digit |
| **isalnum** | *c* is an alphanumeric character |
| **isspace** | *c* is a space, tab, carriage return, newline, or formfeed |
| **ispunct** | *c* is a punctuation character (neither control nor alphanumeric) |
| **isprint** | *c* is a printing character, code 040(8) (space) through 0176 (tilde) |
| **iscntrl** | *c* is a delete character (0177) or ordinary control character (less than 040). |
| **isascii** | *c* is an ASCII character, code less than 0200 |

## 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(1)

## 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 |
| 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** |
| crmode() | set terminal to cbreak mode |
| delch() | delete a character |
| deleteln() | delete a line |
| delwin(win) | delete **win** |
| echo() | set echo mode |
| endwin() | finish up screens |
| 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 |
| insch() | insert a character |
| insertln() | insert a line |
| leaveok(win,boolf) | set leave flag for **win** |
| longname(termbuf,name) | get long name from **termbuf** |

move(y,x) move to (y,x) on **stdscr**
mvcur(lasty,lastx,newy,newx)
actually move cursor
mvscanw(y,x,win,fmt,arg1,arg2...)
move, then do a scanf through the window
mvwin(win,y,x) move position of the window
newwin(lines,cols,begin_y,begin_x)
create a new window
nl() set newline mapping
nocrmode() unset cbreak mode
noecho() unset echo mode
nonl() unset newline mapping
noraw() 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
subwin(lines,cols,begin_y,begin_x)
create a window within a window
touchwin(win) refresh tag for overlapping windows
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**

| | |
|---|---|
| wstandend(win) | unset window in standout mode |
| wstandout(win) | put window in standout mode |

## NAME

ecvt, fcvt, gcvt — output conversion

## SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

## DESCRIPTION

**ecvt** converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

**fcvt** is identical to **ecvt**, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by *ndigit*.

**gcvt** converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in Fortran F-format if possible, otherwise E-format, ready for printing. Trailing zeros may be suppressed.

## SEE ALSO

printf(3)

## BUGS

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

## NAME

end, etext, edata — last locations in program

## SYNOPSIS

**extern end;**
**extern etext;**
**extern edata;**

## DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of **etext** is the first address above the program text, **edata** above the initialized data region, and **end** above the uninitialized data region.

When execution begins, the program break coincides with **end**, but many functions reset the program break, among them the routines of **brk**(2), **malloc**(3), standard input/output (**stdio**(3)), the profile (**-p**) option of **cc**(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see **brk**(2).

## SEE ALSO

brk(2), malloc(3)

## NAME

exp, log, log10, pow, sqrt — exponential, logarithm, power, square root

## SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x, y;

double sqrt(x)
double x;
```

## DESCRIPTION

**exp** returns the exponential function of $x$.

**log** returns the natural logarithm of $x$; **log10** returns the base 10 logarithm of $x$.

**pow** returns $x^y$.

**sqrt** returns the square root of $x$.

## SEE ALSO

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

## DIAGNOSTICS

**exp** and **pow** return a huge value when the correct value would overflow; **errno** is set to ERANGE. **pow** returns 0 and sets **errno** to EDOM when the second argument is negative and non-integral or when both arguments are 0.

**log** returns 0 when $x$ is zero or negative; **errno** is set to EDOM.

**sqrt** returns 0 when $x$ is negative; **errno** is set to EDOM.

## NAME

fclose, fflush — close or flush a stream

## SYNOPSIS

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

## DESCRIPTION

**fclose** causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

**fclose** is performed automatically upon calling **exit**(2).

**fflush** causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

## SEE ALSO

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

## DIAGNOSTICS

These routines return **EOF** if *stream* is not associated with an output file, or if buffered data cannot be transferred to that file.

## NAME

feof, ferror, clearerr, fileno — stream status inquiries

## SYNOPSIS

**#include <stdio.h>**

**feof(stream)**
**FILE *stream;**

**ferror(stream)**
**FILE *stream;**

**clearerr(stream)**
**FILE *stream;**

**fileno(stream)**
**FILE *stream;**

## DESCRIPTION

**feof** returns non-zero when end of file is read on the named input *stream*, otherwise zero.

**ferror** returns non-zero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by **clearerr**, the error indication lasts until the stream is closed.

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

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

These functions are implemented as macros; they cannot be redeclared.

## SEE ALSO

fopen(3), open(2)

## NAME

fabs, floor, ceil — absolute value, floor, ceiling functions

## SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double(x);
```

## DESCRIPTION

**fabs** returns the absolute value $|x|$.

**floor** returns the largest integer not greater than $x$.

**ceil** returns the smallest integer not less than $x$.

## SEE ALSO

abs(3)

## NAME

fopen, freopen, fdopen — open a stream

## SYNOPSIS

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

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

## DESCRIPTION

**fopen** opens the file named by *filename* and associates a stream with it. **fopen** returns a pointer to be used to identify the stream in subsequent operations.

*type* is a character string having one of the following values:

"r" open for reading

"w" create for writing

"a" append: open for writing at end of file, or create for writing

**freopen** substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original stream is closed.

**freopen** reattaches the file pointer *stream* with the file given by filename. *stream* is a value returned by a previous **fopen** or **fdopen** call, or more typically, is the preopened constant name **stdin**, **stdout**, or **stderr**.

**fdopen** associates a stream with a file descriptor obtained from **open**, **dup**, **creat**, or **pipe**(2). The *type* of the stream must agree with the mode of the open file.

## SEE ALSO

open(2), fclose(3)

**DIAGNOSTICS**

**fopen** and **freopen** return the pointer **NULL** if *filename* cannot be accessed.

**BUGS**

**fdopen** is not portable to systems other than UNIX and VENIX.

## NAME

fread, fwrite — buffered binary input/output

## SYNOPSIS

**#include <stdio.h>**

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

**fwrite(ptr, sizeof(*ptr), nitems, stream)**
**FILE *stream;**

## DESCRIPTION

**fread** reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

**fwrite** appends at most *nitems* of data of the type of **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

## SEE ALSO

read(2), write(2), fopen(3), getc(3), putc(3), gets(3), puts(3), printf(3), scanf(3)

## DIAGNOSTICS

**fread** and **fwrite** return 0 upon end of file or error.

## NAME

frexp, ldexp, modf — split into mantissa and exponent

## SYNOPSIS

**double frexp(value, eptr)**
**double value;**
**int *eptr;**

**double ldexp(value, exp)**
**double value;**

**double modf(value, iptr)**
**double value, *iptr;**

## DESCRIPTION

**frexp** returns the mantissa of a double *value* as a double quantity, $x$, of magnitude less than 1 and stores an integer $n$ such that *value* = $x*2**n$ indirectly through *eptr*.

**ldexp** returns the quantity *value**2***exp*.

**modf** returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

## NAME

fseek, ftell, rewind — reposition a stream

## SYNOPSIS

```
#include <stdio.h>

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

long ftell(stream)
FILE *stream;

rewind(stream)
```

## DESCRIPTION

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

**fseek** undoes any effects of **ungetc**(3).

**ftell** returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes on UNIX and VENIX; on some other systems it is a magic cookie, and the only foolproof way to obtain an *offset* for **fseek**.

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

## SEE ALSO

lseek(2), fopen(3)

## DIAGNOSTICS

**fseek** returns −1 for improper seeks.

## NAME

getc, getchar, fgetc, getw — get character or word from stream

## SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

## DESCRIPTION

**getc** returns the next character from the named input *stream*.

**getchar()** is identical to **getc(stdin)**.

**fgetc** behaves like **getc**, but is a genuine function, not a macro; it may be used to save object text.

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

## SEE ALSO

fopen(3), putc(3), gets(3), scanf(3), fread(3), ungetc(3)

## DIAGNOSTICS

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

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

**BUGS**

The end-of-file return from **getchar** is incompatible with that in UNIX editions 1 – 6.

Because it is implemented as a macro, **getc** treats a *stream* argument with side effects incorrectly. In particular, **getc(*f++);** doesn't work sensibly.

## NAME

getenv — value for environment name

## SYNOPSIS

**char *getenv(name)**
**char *name;**

## DESCRIPTION

**getenv** searches the environment list (see **environ**(5)) for a string of the form *name* = *value* and returns *value* if such a string is present, otherwise 0 (NULL).

## SEE ALSO

environ(5), exec(2)

## NAME

getesb, getesw, putesb, putesw — read/write to ES memory

## SYNOPSIS

```
char getesb(addr)
char *addr;

getesw(addr)
int *addr;

putesb(val, addr)
int *addr;
char val;

putesw(val, addr)
int *addr;
int val;
```

## DESCRIPTION

These functions transfer bytes or words between the 8086 Data Segment (DS) and the Extra Segment (ES). The DS is normal user program data space, while the ES is special purpose data space. The **phys** and **sdata**(2) system calls manipulate the ES register, to map it, for example, to the graphics display or to a data area common to several processes. The **getesb()**, **putesb()**, etc. functions then allow a program to **read** and **write** data from and to the extra segment.

**getesb()** takes as an argument a 16 bit address (type ***char**) in the ES and returns the byte value (**char**) at that location. **putesb()** takes two arguments: a 16 bit address (***char**) and a single byte (**char**) value, and places the value at that address in the ES. The functions **getesw()** and **putesw()** provide identical capabilities for word (**int**) transfers.

If a **phys** or **sdata** call has not been done, the ES is identical to the normal DS. In this case, the functions listed here are not likely to be useful, since normal memory to memory transfers are most easily done using standard pointer operations.

## SEE ALSO

phys(2), sdata(2)

## NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent — get group file entry

## SYNOPSIS

```
#include <grp.h>

struct group *getgrent();

struct group *getgrgid(gid) int gid;

struct group *getgrnam(name) char *name;

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 {
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};
```

The members of this structure are:

| | |
|---|---|
| gr_name | The name of the group. |
| gr_passwd | The encrypted password of the group. |
| gr_gid | The numerical group-ID. |
| gr_mem | Null-terminated vector of pointers to the individual member names. |

**getgrent** simply reads the next line while **getgrgid** and **getgrnam** search until a matching *gid* or *name* is found (or until **EOF** is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

A call to **setgrent** has the effect of rewinding the group file to allow repeated searches. **endgrent** may be called to close the group file when processing is complete.

**FILES**

/etc/group

**SEE ALSO**

getlogin(3), getpwent(3), group(4)

**DIAGNOSTICS**

A null pointer (0) is returned on **EOF** or error.

**BUGS**

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

## NAME

getlogin — get login name

## SYNOPSIS

**char *getlogin();**

## DESCRIPTION

**getlogin** returns a pointer to the login name as found in **/etc/utmp**. It may be used in conjunction with **getpwnam** to locate the correct password file entry when the same userid is shared by several login names.

If **getlogin** is called within a process that is not attached to a terminal, it returns **NULL**. The correct procedure for determining the login name is to first call **getlogin** and if it fails, to call **getpwuid**.

## FILES

/etc/utmp

## SEE ALSO

getpwent(3), getgrent(3), utmp(4)

## DIAGNOSTICS

Returns **NULL** (0) if name not found.

## BUGS

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

## NAME

getpass — read a password

## SYNOPSIS

**char *getpass(prompt)**
**char *prompt;**

## DESCRIPTION

**getpass** reads a password from the file **/dev/tty**, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

## FILES

/dev/tty

## SEE ALSO

crypt(3)

## BUGS

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

## NAME

getpw — get name from UID

## SYNOPSIS

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

## DESCRIPTION

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

## FILES

/etc/passwd

## SEE ALSO

getpwent(3), passwd(4)

## DIAGNOSTICS

Non-zero return on error.

## NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent — get password file entry

## SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent();

struct passwd *getpwuid(uid) int uid;

struct passwd *getpwnam(name) char *name;

int setpwent();

int endpwent();
```

## DESCRIPTION

**getpwent**, **getpwuid**, and **getpwnam** each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

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

The fields *pw_quota* and *pw_comment* are unused; the others have meanings described in **passwd**(4).

**getpwent** reads the next line (opening the file if necessary); **setpwent** rewinds the file; **endpwent** closes it.

**getpwuid** and **getpwnam** search from the beginning until a matching *uid* or *name* is found (or until **EOF** is encountered).

**FILES**

/etc/passwd

**SEE ALSO**

getlogin(3), getgrent(3), passwd(4)

**DIAGNOSTICS**

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

**BUGS**

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

## NAME

gets, fgets — get a string from a stream

## SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

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

## DESCRIPTION

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

**fgets** reads $n-1$ characters, or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. **fgets** returns its first argument.

## SEE ALSO

puts(3), getc(3), scanf(3), fread(3), ferror(3)

## DIAGNOSTICS

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

## BUGS

**gets** deletes a newline, **fgets** keeps it, all in the name of backward compatibility.

## NAME

hypot, cabs — euclidean distance

## SYNOPSIS

```
#include <math.h>

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

double cabs(z)
struct { double x, y;} z;
```

## DESCRIPTION

**hypot** and **cabs** return

$$\mathbf{sqrt}(x*x + y*y),$$

taking precautions against unwarranted overflows.

## SEE ALSO

exp(3) for **sqrt**

## NAME

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

## SYNOPSIS

```
#include <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)
double x;
```

## DESCRIPTION

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

## DIAGNOSTICS

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

## NAME

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

## SYNOPSIS

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

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

## DESCRIPTION

**l3tol** converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

**ltol3** performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

**NAME**

libpc — library of external routines for Pascal programs

**SYNOPSIS**

```
const   bufsize = ?;
type    br1 = 1..bufsize;
        br2 = 0..bufsize;
        br3 = -1..bufsize;
        ok = -1..0;
        buf = packed array[br1] of char;
        alfa = packed array[1..8] of char;
        string = packed array[1..?] of char;
        filetype = file of ?;
        long = record high,low:integer end;

{all routines must be declared extern}

function argc:integer;
function argv(i:integer):string;
function environ(i:integer):string;
procedure       argshift;

procedure       buff(var f:filetype);
procedure       nobuff(var f:filetype);
procedure       notext(var f:text);
procedure       diag(var f:text);
procedure       pcreat(var f:text; s:string);
procedure       popen(var f:text; s:string);
procedure       pclose(var f:filetype);

procedure       trap(err:integer);
procedure       encaps(procedure p; procedure q(n:integer));

function perrno:integer;
function uread(fd:integer; var b:buf; len:br1):br3;
function uwrite(fd:integer; var b:buf; len:br1):br3;

function strbuf(var b:buf):string;
function strtobuf(s:string; var b:buf; len:br1):br2;
function strlen(s:string):integer;
function strfetch(s:string; i:integer):char;
procedure       strstore(s:string; i:integer; c:char);
function clock; integer;
```

## DESCRIPTION

This library contains some often used external routines for Pascal programs. Two versions exist: one for the EM-1 interpreter and another one that is used when programs are translated into PDP-11 code. The routines can be divided into several categories:

Argument control:

argc — Gives the number of arguments provided when the program is called.

argv — Selects the specified argument from the argument list and returns a pointer to it. This pointer is nil if the index is out of bounds ($<0$ or $>=$ argc).

environ — Returns a pointer to the i-th environment string ($i>=0$). Returns null if i is beyond the end of the environment list.

argshift — Effectively deletes the first argument from the argument list. Its function is equivalent to 'shift' in the VENIX shell: argv[2] becomes argv[1], argv[3] becomes argv[2], etc. It is a useful procedure to skip optional flag arguments. Note that the matching of arguments and files is done at the time a file is opened by a call to reset or rewrite.

Additional file handling routines:

buff — Turn on buffering of a file. Not very useful, because all files are buffered except standard output to a terminal and diagnostic output. Input files are always buffered.

nobuff — Turn off buffering of an output file. It causes the current contents of the buffer to be flushed.

notext — Only useful for input files. End of line characters are not replaced by a space and character codes out of the ASCII range ($0-127$) do not cause an error message.

diag — Initialize a file for output on the diagnostic output stream ($fd=2$). Output is not buffered.

pcreat — The same as rewrite(f), except that you must provide the filename yourself. The name must be zero terminated. Only text files are allowed.

popen — The same as reset(f), except that you must provide the filename yourself. The name must be zero terminated. Only text files are allowed.

pclose — Gives you the opportunity to close files hidden in records or arrays. All other files are closed automatically.

String handling:

strbuf — Type conversion from character array to string. It is your own responsibility that the string is zero terminated.

strtobuf — Copy string into buffer until the string terminating zero byte is found or until the buffer if full, whatever comes first. The zero byte is also copied. The number of copied characters, excluding the zero byte, is returned. So if the result is equal to the buffer length, then the end of buffer is reached before the end of string.

strlen — Returns the string length excluding the terminating zero byte.

strfetch — Fetches the i-th character from a string. There is no check against the string length.

strstore — Stores a character in a string. There is no check against string length, so this is a dangerous procedure.

Trap handling:

These routines allow you to handle all the possible error situations yourself. You may define your own trap handler, written in Pascal, instead of the default handler that produces an error message and quits. You may also generate traps yourself.

trap — Trap generates the trap passed as argument (0 – 255). The trap numbers 128 – 255 may be used freely. The others are reserved for standard run-time errors.

encaps — Encapsulate the execution of 'p' with the trap handler 'q'. Encaps replaces the previous trap handler by 'q', calls 'p' and restores the previous handler when 'p' returns. If, during the execution of 'p', a trap occurs, then 'q' is called with the trap number as

parameter. For the duration of 'q' the previous trap handler is restored, so that you may handle only some of the errors in 'q'. All the other errors must then be raised again by a call to 'trap'.

Encapsulations may be nested: you may encapsulate a procedure while executing an encapsulated routine.

Jumping out of an encapsulated procedure (non-local goto) is dangerous, because the previous trap handler must be restored. Therefore, you may only jump out of procedure 'p' from inside 'q' and you may only jump out of one level of encapsulation. If you want to exit several levels of encapsulation, use traps. Note that 'p' may not have parameters.

The following error codes are used by the Pascal runtime system:

64 more args expected
65 error in exp
66 error in ln
67 error in sqrt
68 assertion failed
69 array bound error in pack
70 array bound error in unpack
71 only positive j in 'i mod j'
72 file not yet open
96 file xxx: not writable
97 file xxx: not readable
98 file xxx: end of file
99 file xxx: truncated
100 file xxx: reset error
101 file xxx: rewrite error
102 file xxx: close error
103 file xxx: read error
104 file xxx: write error
105 file xxx: digit expected
106 file xxx: non-ASCII char read

VENIX system calls:

The routines of this category require global variables or routines of the monitor library **libmon**(3).

uread — Equal to the read system call. Its normal name is blocked by the standard Pascal routine read.

uwrite — As above but for **write**(2).

perrno — Because external data references are not possible in Pascal, this routine returns the global variable errno, indicating the result of the last system call.

Miscellaneous:

clock — Return the number of ticks of user and system time consumed by the program.

## FILES

/usr/lib/libpc.a — the version for compiled programs

/usr/lib/em1_pc.a — the version for interpreted programs

## SEE ALSO

pc(1), pc_emlib(3), pc_prlib(3), libmon(3)

## DIAGNOSTICS

Two routines may cause fatal error messages to be generated. These are:

pcreat — Rewrite error (trap 77) if the file cannot be created.

popen — Reset error (trap 76) if the file cannot be opened for reading

## AUTHOR

Johan Stevenson, Vrije Universiteit.

## NAME

malloc, free, realloc, calloc — main memory allocator

## SYNOPSIS

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

## DESCRIPTION

**malloc** and **free** provide a simple general-purpose memory allocation package. **malloc** returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to **free** is a pointer to a block previously allocated by **malloc**; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by **malloc** is overrun or if some random number is handed to **free**.

**malloc** allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls **sbrk** (see **break**(2)) to get more memory from the system when there is no suitable space already free.

**realloc** changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

**realloc** also works if *ptr* points to a block freed since the last call of **malloc**, **realloc**, or **calloc**; thus sequences of **free**, **malloc**, and **realloc** can exploit the search strategy of **malloc** to do storage compaction.

**calloc** allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

## DIAGNOSTICS

**malloc**, **realloc**, and **calloc** return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

## BUGS

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 X's, which will be replaced with the current process id and a unique letter.

## SEE ALSO

getpid(2)

## NAME

monitor — prepare execution profile

## SYNOPSIS

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

## DESCRIPTION

An executable program created by '**cc −p**' automatically includes calls for **monitor** with default parameters; **monitor** needn't be called explicitly except to gain fine control over profiling.

**monitor** is an interface to **profil**(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. **monitor** arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **−p** of **cc**(1) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext( );
...
monitor((int)2, etext, buf, bufsize, nfunc);
```

**etext** lies just above all the program text, see **end**(3).

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor(0);
```

then **prof**(1) can be used to examine the results.

## FILES

mon.out

## SEE ALSO

prof(1), profil(2), cc(1)

## NAME

itom, madd, msub, mult, mdiv, min, mout, pow, gcd, rpow — multiple precision integer arithmetic

## SYNOPSIS

**typedef struct { int len; short *val; } mint;**

**madd(a, b, c)**
**msub(a, b, c)**
**mult(a, b, c)**
**mdiv(a, b, q, r)**
**min(a)**
**mout(a)**
**pow(a, b, m, c)**
**gcd(a, b, c)**
**rpow(a, b, c)**
**msqrt(a, b, r)**
**mint *a, *b, *c, *m, *q, *r;**

**sdiv(a, n, q, r)**
**mint *a, *q;**
**short *r;**

**mint *itom(n)**

## DESCRIPTION

These routines perform arithmetic on integers of arbitrary length. The integers are stored using the defined type **mint.** Pointers to a **mint** should be initialized using the function **itom**, which sets the initial value to *n*. After that space is managed automatically by the routines.

**madd**, **msub**, **mult**, assign to their third arguments the sum, difference, and product, respectively, of their first two arguments. **mdiv** assigns the quotient and remainder, respectively, to its third and fourth arguments. **sdiv** is like **mdiv** except that the divisor is an ordinary integer. **msqrt** produces the square root and remainder of its first argument. **rpow** calculates *a* raised to the power *b*, while **pow** calculates this reduced modulo *m*. **min** and **mout** do decimal input and output.

The functions are obtained with the loader option **−lmp**.

**DIAGNOSTICS**

Illegal operations and running out of memory produce messages and core images.

## NAME

nlist — get entries from name list

## SYNOPSIS

```
#include <a.out.h>
nlist(filename, nl)
char *filename;
struct nlist nl[ ];
```

## DESCRIPTION

**nlist** examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See **a.out**(4) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file **/venix**. In this way programs can obtain system addresses that are up to date.

## SEE ALSO

a.out(4)

## DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

**NAME**

pc_prlib — library of Pascal runtime routines

**SYNOPSIS**

```
type        alpha = packed array[1..8] of char;
            pstring = ^packed array[] of char;

function    _abi(i:integer):integer;
function    _abl(i:long):long;
function    _mdi(i,j:integer):integer;
function    _mdl(i,j:long):long;
function    _abr(r:real):real;
function    _sin(r:real):real;
function    _cos(r:real):real;
function    _atn(r:real):real;
function    _exp(r:real):real;
function    _log(r:real):real;
function    _sqt(r:real):real;
function    _rnd(r:real):integer;

type        compared = -1..1;
            gotoinfo = record
               procdesc:integer;
               pcoffset:integer;
               nlocals: integer;
            end;

function    _bcp(s1,s2:pstring; sz:integer):compared;
function    _bts(low,high,size:integer):set of 0..(8*size-1);
procedure   _gto(p:^gotoinfo);

procedure   _new(var p:^integer; size:integer);
procedure   _dis(var p:^integer; size:integer);
procedure   _sav(var p:^integer);
procedure   _rst(var p:^integer);

type        arrdescr = record
               lowbnd:  integer;
               diffbnds:integer;
               elsize:  integer;
            end;
            arr1 = array[] of ?;
            arr2 = packed array[] of ?;
```

```
procedure   _pac(var ap:arr1; i:integer; var zp:arr2;
                 var zd,ad:arrdescr);
procedure   _unp(var zp:arr2; var ap:arr1; i:integer;
                 var zd,ad:arrdescr);
function    _asz(var dp:arrdescr):integer;

procedure   _ass(b:boolean; line:integer);
procedure   procentry(var name:alpha);
procedure   procexit(var name:alpha);

const       lowbyte = [0..7];
            MAGIC  = [1,3,5,7];
            WINDOW = [11];
            ELNBIT = [12];
            EOFBIT = [13];
            TXTBIT = [14];
            WRBIT  = [15];
type        file = record
                ptr:    ^char;
                flags:  set of [0..15];
                fname:  string;
                ufd:    0..15;
                size:   integer;
                count:  0..buflen;
                buflen: max(512,size) div size * size;
                bufadr: packed array[1..max(512,size)]
                            of char;
            end;
            filep = ^file;
const       NFILES = 15;
            _extfl:^array[ ] of filep;

procedure   _ini(var p:array[ ] of filep);
procedure   _hlt(status:0..255);

procedure   _opn(f:filep; size:integer);
procedure   _cre(f:filep; size:integer);
procedure   _cls(f:filep);

procedure   _get(f:filep);
procedure   _put(f:filep);
function    _wdw(f:filep):^char;
function    _efl(f:filep):boolean;
```

```
function    _eln(f:filep):boolean;
function    _rdc(f:filep):char;
function    _rdi(f:filep):integer;
function    _rdl(f:filep):long;
function    _rdr(f:filep):real;
procedure   _rln(f:filep);
procedure   _wrc(f:filep; c:char);
procedure   _wsc(f:filep; c:char; w:integer);
procedure   _wri(f:filep; i:integer);
procedure   _wsi(f:filep; i:integer; w:integer);
procedure   _wrl(f:filep; l:long);
procedure   _wsl(f:filep; l:long; w:integer);
procedure   _wrr(f:filep; r:real);
procedure   _wsr(f:filep; r:real; w:integer);
procedure   _wrf(f:filep; r:real; w:integer; ndigit:integer);
procedure   _wrs(f:filep; s:pstring; l:integer);
procedure   _wss(f:filep; s:pstring; l:integer; w:integer);
procedure   _wrb(f:filep; b:boolean);
procedure   _wsb(f:filep; b:boolean; w:integer);
procedure   _wrz(f:filep; s:string);
procedure   _wsz(f:filep; s:string; w:integer);
procedure   _wln(f:filep);
procedure   _pag(f:filep);
```

**DESCRIPTION**

This library is used by the Pascal compiler. Two versions exist: one for use with interpretive (EM-1) code, and the other for compiled PDP-11 code. This library contains all the runtime routines for standard Pascal programs. These routines can be divided into several categories. A description of each category with its routines follows.

Arithmetic routines:

| | |
|---|---|
| _abi | Compute the absolute value of an integer. |
| _abl | Compute the absolute value of a long. |
| _mdi | Perform the Pascal modulo operation on integers. |
| _mdl | Perform the Pascal modulo operation on longs. |
| _abr | Compute the absolute value of a real. |
| _sin | Compute the sine of a real. |
| _cos | Compute the cosine of a real. |
| _atn | Compute the arc tangent of a real. |

_exp Compute the e-power of a real.

_log Compute the natural logarithm of a real.

_sqt Compute the square root of a real.

_rnd Round a real to the nearest integer ($-3.5 \rightarrow -4$).

Miscellaneous routines:

_bcp Compare two strings. Use dictionary ordering with the ASCII character set.

_bts Include a range of elements from low to high in a set of size bytes (size is even).

_gto Execute a non-local goto. When called, the static link points to the local base of the target procedure. The new EM-1 stack pointer is calculated by adding the number of locals to the new local base (jumping into statements is not allowed; there are no local generators in Pascal!). The new program counter can be computed out of the procedure descriptor number and the program counter offset.

Heap management:

There is one way to allocate new heap space (_new), but two different incompatible ways to deallocate it.

The most general one is by using dispose (_dis). A circular list of free blocks, ordered from low to high addresses, is maintained. Merging free blocks is done when a new block enters the free list. When a new block is requested (_new), the free list is searched using a first fit algorithm. Two global variables are needed:

_highp Points to the free block with the highest address.

_lastp Points to the most recently entered free block or to a block in the neighborhood of the most recently allocated block. The free list is empty, when one of these pointers (but then at the same time both) is zero.

The second way to deallocate heap space is by using mark (_sav) and release (_rst). Mark saves the current value of the heap pointer HP in the program variable passed as a parameter. By

calling release with this old HP value as its argument, the old HP value is restored, effectively deallocating all blocks requested between the calls to mark and release. The heap is used as second stack in this case.

It will be clear that these two ways of deallocating heap space can not be used together. To be able to maintain the free list, all blocks must be a multiple of 4 bytes long, with a minimum of 4 bytes.

In summary:

_new Allocate heap space.

_dis Deallocate heap space.

_sav Save the current value of HP.

_rst Restore an old value of HP.

Array operations:

The only useful form of packing implemented, is packing words into bytes. All other forms of packing and unpacking result in a plain copy.

_pac Pack an unpacked array 'a' into a packed array 'z'. 'ap' and 'zp' are pointers to 'a' and 'z'. 'ad' and 'zd' are pointers to the descriptors of 'a' and 'z'. 'i' is the index in 'a' of the first element to be packed. Pack until 'z' is full.

_unp Unpack 'z' into 'a'. 'ap', 'zp', 'ad' and 'zd' are as for _pac. 'i' is the index in 'a' where the first element of 'z' is copied into. Unpack all elements of 'z'.

_asz Compute array size. Used for copying conformant arrays.

Debugging facilities:

The compiler allows you to verify assertions. It generates a call to the routine _ass to check the assertion at runtime. Another feature of the compiler is that it enables you to trace the procedure calling sequence. If the correct option is turned on, then a call to the procedure 'procentry' is generated at the start of each compiled procedure or function. Likewise, the routine 'procexit'

is called just before a procedure or function exits. Default procedure 'procentry' and 'procexit' are available in this library.

_ass — If 'b' is zero, then change eb[0] to 'line' (to give an error message with source line number) and call the error routine.

procentry — Print the name of the called procedure with up to seven argument words in decimal on standard output. Output must be declared in the program heading.

procexit — Print the name of the procedure that is about to exit. Same remarks as for procentry.

Files:

Most of the runtime routines are needed for file handling. For each file in your Pascal program a record of type file, as described above, is allocated, static if your file is declared in the outermost block, dynamic if it is declared in inner blocks. The fields in the file record are used for:

bufadr — IO is buffered except for standard input and output if terminals are involved. The size of the buffer is the maximum of 512 and the file element size.

buflen — The effective buffer length is the maximum number of file elements fitting in the buffer, multiplied by the element size.

size — The file element size (1 or even).

flags — Some flag bits are stored in the high byte and a magic pattern in the low byte provides detection of destroyed file information.

ptr — Points to the file window inside the buffer.

count — The number of bytes (the window inclusive) left in the buffer to be read or the number of free bytes (the window inclusive) for output files.

ufd — The VENIX file descriptor for the file.

fname — Points to the name of the file (INPUT for standard input, OUTPUT for standard output and LOCAL for local files). This field is used for generating error messages.

The constants used by the file handling routines are:

WINDOW
: Bit in flags set if the window of an input file is initialized. Used to resolve the famous interactive input problem.

EOFBIT
: Bit in flags set if end of file seen

ELNBIT
: Bit in flags set if linefeed seen

TXTBIT
: Bit in flags set for text files. Process linefeeds.

WRBIT
: Bit in flags set for output files

MAGIC
: Pattern for the low byte of flags

NFILES
: The maximum number of open files in VENIX

Prelude and postlude:

These routines are called once for each Pascal program:

_ini
: When a file mentioned in the program heading is opened by reset or rewrite, its file pointer must be mapped onto one of the program arguments. The compiler knows how to map and therefore builds a table with a pointer to the file structure for each program argument. One of the first actions of the Pascal program is to call this procedure with this table as an argument. The global variable _extfl is used to save the address of this table. Another task of _ini is to initialize the standard input and output files. For standard output it must decide whether to buffer or not. If standard output is a terminal, then buffering is off by setting buflen to 1. A last task of _ini is to set the global variables _argc, _argv and _environ for possible reference later on.

_hlt
: If the program is about to finish, the buffered files must be flushed. That is done by this procedure.

Opening and closing:

Files in Pascal are opened for reading by reset and opened for writing by rewrite. Files to be rewritten may or may not exist already. Files not mentioned in the program heading are considered local files. The next steps must be done for reset and rewrite:

1. If size is zero, then a text file must be opened with elements of size 1.
2. Find out if this file is mentioned in the program heading (scan table pointed to by _extfl). If not, then it is a local file and goto 7.
3. If the file is standard input or output then return.
4. If there are not enough arguments supplied, generate an error.
5. If the file was already open, flush the buffer if necessary and close it. Note that reset may be used to force the buffer to be flushed. This is sometimes helpful against program or system crashes.
6. If it is a reset, open the file, otherwise create it. In both cases goto 9.
7. If the local file is to be written, then close it if it was open and create a new nameless file. First try to create it in /usr/tmp, then in /tmp and if both fail then try the current directory. See to it that the file is open for both reading and writing.
8. If the local file is to be read and the file is opened already, then flush the buffer and seek to the beginning. Otherwise open a temporary file as described in 7.
9. Initialize all the file record fields.

The necessary procedures are:

_opn Reset a file

_cre Rewrite a file

_cls Close a file. Closing of files is done for local files when the procedure in which they are declared exits. The compiler only closes local files if they are not part of a structured type. Files allocated in the heap are not closed when they are deallocated. There is an external routine 'pclose' in **libpc**(3), that may be called explicitly to do the closing in these cases. Closing may be necessary to flush buffers or to keep the number of simultaneously opened files below NFILES. Files declared in the outermost block are automatically closed when the program terminates.

General file IO:

These routines are provided for general file IO:

_put Append the file element in the window to the file and advance the window.

_get Advance the file window so that it points to the next element of the file. For text files (TXTBIT on) the ELNBIT in flags is set if the new character in the window is a line feed (ASCII 10) and the character is then changed into a space. Otherwise the ELNBIT is cleared.

_wdw Return the current pointer to the file window.

_eof Test if you reached end of file. Is always true for output files.

Textfile routines:

The rest of the routines all handle text files.

_eln Return true if the next character on an input file is an end-of-line marker. An error occurs if eof(f) is true.

_rdc Return the character currently in the window and advance the window.

_rdi Build an integer from the next couple of characters on the file, starting with the character in the window. The integer may be preceded by spaces (and line feeds), tabs and a sign. There must be at least one digit. The first non-digit signals the end of the integer.

_rdl Like _rdi, but for longs.

_rdr Like _rdi, but for reals. Syntax is as required for Pascal.

_rln Skips the current line and clears the WINDOW flag, so that the next routine requiring an initialized window knows that it has to fetch the next character first.

_wrc Write a character, not preceeded by spaces.

_wsc Write a character, left padded with spaces up to a field width of 'w'.

_wri Write an integer, left padded with spaces up to a field width of 6.

_wsi Write an integer, left padded with spaces up to a field width of 'w'.

_wrl Write a long, left padded with spaces up to a field width of 11.

_wsl Write a long, left padded with spaces up to a field width of 'w'.

_wrr Write a real in scientific format, left padded with spaces up to a field width of 13.

_wsr Write a real in scientific format, left padded with spaces up to a field width of 'w'.

_wrf Write a real in fixed point format, with exactly 'ndigit' digits behind the decimal point, the last one rounded; it is left padded up to a field width of 'w'.

_wrs Write a string of length 'l', without additional spaces.

_wss Write a string of length 'l', left padded up to a field width of 'w'.

_wrb Write a boolean, represented by "true" or "false", left padded up to a field width of 5.

_wsb Write a boolean, represented by "true" or "false", left padded up to a field width of 'w'.

_wrz Write a C-type string up to the zero-byte.

_wsz Write a C-type string, left padded up to a field width of w.

_wln Write a line feed (ASCII 10).

_pag Write a form feed (ASCII 12).

All the routines to which calls are generated by the compiler are described above. They use the following global defined routines to do some of the work:

_rf Check input files for MAGIC and WRBIT. Initialize the window if WINDOW is cleared.

_wf Check output files for MAGIC and WRBIT.

_incpt Advance the file window and read a new buffer if necessary.

_outcpt Write out the current buffer if necessary and advance the window.

_flush Flush the buffer if it is an output file. Append an extra line marker if EOLBIT is off.

_wstrin All output routines make up a string in a local buffer. They call _wstrin to output this buffer and to do the left padding.

_skipsp Skip spaces (and line feeds) on input files.

_getsig Read '+' or '−' if present.

_fstdig See to it that the next character is a digit. Otherwise error.

_nxtdig Check if the next character is a digit.

_getint Do the work for _rdi.

_ecvt Convert real into string of digits for printout in scientific notation.

| | |
|---|---|
| _fcvt | Convert real into string of digits for fixed point printout |
| _fif | Split real into integer and fraction part |
| _fef | Split real into exponent and fraction part |

The following global variables are used:

| | |
|---|---|
| _lastp | For heap management (see above). |
| _highp | For heap management (see above). |
| _extfl | Used to save the argument of _ini for later reference. |
| _curfil | Save the current file pointer, so that the error message can access the file name. |

**FILES**

| | |
|---|---|
| /lib/pc_prlib.a | The library used by compiled programs. |
| /lib/em1_pr.a | The library used by interpreted programs. |
| /lib/pc/rterrors | The error messages |

**SEE ALSO**

[1] A.S. Tanenbaum, J.W. Stevenson & Hans van Staveren "Description of an experimental machine architecture for use of block structured languages" Informatica rapport IR – 54.

[2] K.Jensen & N.Wirth "PASCAL, User Manual and Report" Springer-Verlag.

[3] An improved version of the ISO standard proposal for the language Pascal ISO/TC97/SC5 – N462, received November 1979.

[4] "VU-Pascal Reference Manual" in the *Programming Guide*.

[5] pc(1)

**DIAGNOSTICS**

All errors discovered by this runtime system cause an EM-1 TRP instruction to be executed. This TRP instruction expects the error number on top of the stack. See [1] for a more extensive treatment of the subject.

EM-1 allows the user to specify a trap handling routine, called whenever an EM-1 machine trap or a language or user defined trap occurs. One of the first actions in _ini is to specify that the routine _fatal, available in this library, will handle traps. This routine is called with an error code (0..255) as argument. The file "/lib/pc/rterrors" is opened and searched for a message corresponding with this number. If the file can not be opened, or if the error number is not recorded in the file, then the same trap is generated again, but without a user-defined trap handler, so that

the low levels generate an error message. Otherwise the following information is printed on file descriptor 2:

- The name of the Pascal program
- The name of the file pointed to by _curfil, if the error number is between 96 and 127 inclusive.
- The error message (or the error number if not found).
- The source line number if not equal to 0.

The routine _fatal stops the program as soon as the message is printed.

The following error codes are used by the Pascal runtime system:

| | |
|---|---|
| 64 | more args expected |
| 65 | error in exp |
| 66 | error in ln |
| 67 | error in sqrt |
| 68 | assertion failed |
| 69 | array bound error in pack |
| 70 | array bound error in unpack |
| 71 | only positive j in 'i mod j' |
| 72 | file not yet open |
| | |
| 96 | file xxx: not writable |
| 97 | file xxx: not readable |
| 98 | file xxx: end of file |
| 99 | file xxx: truncated |
| 100 | file xxx: reset error |
| 101 | file xxx: rewrite error |
| 102 | file xxx: close error |
| 103 | file xxx: read error |
| 104 | file xxx: write error |
| 105 | file xxx: digit expected |
| 106 | file xxx: non-ASCII char read |

**AUTHORS**

Johan Stevenson and Ard Verhoog, Vrije Universiteit.

## NAME

perror, sys_errlist, sys_nerr — system error messages

## SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[ ];
```

## DESCRIPTION

**perror** produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable **errno** (see **intro**(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings **sys_errlist** is provided; **errno** can be used as an index in this table to get the message string without the newline. **sys_nerr** is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

## SEE ALSO

intro(2)

## NAME

openpl *et al.* — graphics interface

## SYNOPSIS

Standard Unix subroutines

| | |
|---|---|
| **openpl()** | |
| **erase()** | |
| **space(x0, y0, x1, y1)** | int x0, y0, x1, y1; |
| **label(string)** | char string[ ]; |
| **line(x0, y0, x1, y1)** | int x0, y0, x1, y1; |
| **circle(x, y, rad)** | int x, y, rad; |
| **arc(x, y, x0, y0, x1, y1)** | int x, y, x0, y0, x1, y1; |
| **move(x, y)** | int x, y; |
| **cont(x, y)** | int x, y; |
| **point(x, y)** | int x, y; |
| **linemod(style)** | char style[ ]; |
| **closepl()** | |

Special VENIX Enhancements

| | |
|---|---|
| **linepat(pat)** | int pat; |
| **linewid(width)** | int width; |
| **window(x0, y0, x1, y1)** | int x0, y0, x1, y1; |
| **box(x0, y0, x1, y1)** | int x0, y0, x1, y1; |
| **rfill(x0, y0, x1, y1)** | int x0, y0, x1, y1; |
| **fill(x, y)** | int x, y; |
| **dot(x, y, rad)** | int x, y, rad; |
| **color(col)** | int col; |
| **colndx(col, pat)** | int col, pat; |
| **writemod(s)** | char s[ ]; |

## DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. **openpl()** must be used before any of the others to open the device for graphics. **closepl()** flushes the output and closes the device.

String arguments to **label()**, **linemod()**, and **writemod()** are null-terminated, and do not contain newlines.

All coordinate points used in the routines are user-coordinates, defined by the **space()** subroutine. **space()** must be called to set up this coordinate system, or you may get strange results!

The last designated point in a call to **line()**, **move()**, **cont()**, or **point()**, becomes the 'current point' for the next plotting instruction.

Standard Unix subroutines

**openpl()**
Initialize graphics output device for writing. The routine will return a zero value upon a successful initialization. If an error is encountered, the routine will return a value of −1.

**erase()** Erase the graphics screen.

**space(***x0, y0, x1, y1***)**
Set up plotting area. The coordinate points (x0,y0) and (x1,y1) specify respectively the lower-left and upper-right corners of the user-coordinate system to be defined. The user-coordinate system is scaled to fit the largest possible square region allowable on the graphics device screen. All plotting instructions are produced with respect to the user-coordinate system.
The upper limits defined are just outside the plotting area.
Video screens which are not square will display a blank portion outside the plotting area. You may plot beyond the space settings in order to take advantage of this area.

**label(***s***)** Write out the ASCII text string so that its first character falls on the current point.

**line(***x0, y0, x1, y1***)**
Draw a line from (x0,y0) to (x1,y1). (x1,y1) becomes the new current point.

**circle(***x, y, r***)**
Draw a circle with center at (x,y) having radius r.

**arc(***x, y, x0, y0, x1, y1***)**
Draw an arc with center at (x,y). The next two points determine the starting and ending octants (inclusive and exclusive) for a counter-clockwise arc. The point (x0,y0) is referenced for arc radius.

**move(***x, y***)**
Move the current point to (x,y).

**cont(*x, y*)**
Draw from current point to (x,y). (x,y) becomes the new current point.

**point(*x, y*)**
Draw a point at (x,y). (x,y) becomes the new current point.

**linemod(*s*)**
Change line-style attribute for subsequent lines. Styles available are: 'solid', 'dotted', 'longdashed', 'shortdashed', 'dotdashed'.

**closepl()**
Close graphics output device for writing.

Special VENIX Enhancements

**linepat(*pattern*)**
Specify a 16-bit integer bit pattern, of user's choice, to be used in place of one of the available line styles. Bits set to '1' are visible; bits set to '0' are invisible.
EX: The integer *0146314* in octal represents *'1100110011001100'* which is a dashed line.

**linewid(*width*)**
Specify line width in user coordinates.

**window(*x0, y0, x1, y1*)**
The points (x0,y0) and (x1,y1) specify respectively the lower-left and upper-right corners of a clipping window in user coordinates. Only portions of lines inside the window will be displayed, and all erasures will be confined to the window area only.

**box(*x0, y0, x1, y1*)**
Draw a box frame bounded by the lower-left and upper-right points (x0,y0) and (x1,y1).

**rfill(*x0, y0, x1, y1*)**
Draw a filled rectangle bounded by the user coordinates (x0,y0) and (x1,y1). The fill pattern is determined by the current line-style pattern (see **linepat()**).

**fill(*x, y*)**
Fill a convex closed boundary of arbitrary shape. The point (x,y) is the seed point (starting point) and must be

inside the boundary. The fill pattern is determined by the current line-style pattern (see **linepat()**).

**dot(***x, y, r***)**

Draw a filled circle with center at (x,y), having a radius r. The fill pattern is determined by the current line-style pattern (see **linepat()**).

**color(***c***)**

Choose a color from the present color palette. All graphics following will be displayed in that color. The color palette is preset as follows:

| color | shade |
|---|---|
| 0 | black (background) |
| 1 | blue |
| 2 | green |
| 3 | cyan |
| 4 | red |
| 5 | magenta |
| 6 | yellow |
| 7 | white |

The **colndx()** routine below explains how to change the color palette. On a monochrome monitor, the colors will show up as gray scales. Be cautious, as this routine *should not* be called unless an extended bit map board (color board) is present in the machine.

**colndx(***c, pat***)**

Set a color from the palette to a desired shade. In the DEC PRO color graphics, the colors available are numbered 0 thru 7, 0 being the background color. The pattern argument is a 16-bit integer to be set by the user.

| 15 – – 8 | 7 – – 0 |
|---|---|
| 0 ... 0 | r r r g g g b b |

The high 8 bits are ignored and should be set to zero. The low 8 bits specify the individual intensity of the three primary colors: red, green, and blue. Intensities for red and green may range from 0 to 7, and blue from 0 to 3. For example, a pattern of *'034'* in octal represents *'00000000 000 111 00'* in binary, which is a full-intensity green.

Be cautious, as a change in the color palette will instantly change that color on the screen. It is advised that this routine be called before any plotting to eliminate color flashing. This may be useful, however, for creating special effects for demo purposes. *Do not* use this routine unless an extended bit map board (color board) is present in the machine.

**writemod(*s*)**

Choose from one of the five available writing modes:

**xor** *Exclusive-or* mode allows one to overlay several images onto the same screen, and remove them arbitrarily while retaining the underlying image.
[exclusive 'OR' data to screen: memory ^= data]

**mov** *Absolute-move* mode overwrites anything on the screen and is good for clearing off previous images.
[move data to screen: memory = data]

**mvc** *Move-complement* mode overwrites the screen with a reverse image, creating reverse-video effects.
[move complement of data to screen: memory = ~data]

**bis** *Bit-set* mode writes only the set (turned-on) bits onto the screen. The current image is not destroyed, and thus this mode is useful for creating composite images.
['OR' data to screen: memory |= data]

**bic** *Bit-clear* mode writes only the clear (turned-off) bits onto the screen.
['AND' complement of data to screen: memory &= ~data]

All plotting instructions operate in the chosen writing mode. *Bit-set* mode is the default.

Various flavors of these functions exist for different output devices. They are obtained by the following **ld**(1) options (normally placed at the end of the line in the **cc** command):

**-lplot** device-independent graphics stream on standard output for **plot**(1g) filters

The following options bypass the plot filters and write directly to the device:

**-ltpro** DEC PRO monochrome and color graphics screens
Color is distinguished from monochrome by a first subroutine call to **color()**. If you want monochrome output, do not make any calls to either **color()** or **colndx()**. Color will work only if an extended bit map board (color board) is present in the machine. If your machine has a monchrome monitor *and* a color board, use of the color subroutines will produce gray scales.

**-lt4014** Tektronix 4014 terminal

## EXAMPLES

To make a program using the device independent plot library:

```
cc -o program program.c -lplot
```

To run the program on the DEC PRO graphics screen:

```
program < data | plot -Tpro
```

To run the program on a Tektronix terminal:

```
program < data | plot -T4014 > /dev/tek
```

To bypass the plot filters and write directly to the DEC PRO:

```
cc -o program program.c -ltpro
program < data
```

## SEE ALSO

plot(1g), plot(5), setscreen(1g)

## NOTES

The **fill()** routine operates a boundary-fill operation and requires an unbroken boundary in order to fill properly. Please be cautious when using it.

## NAME

popen, pclose — initiate I/O to/from a process

## SYNOPSIS

```
#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;
```

## DESCRIPTION

The arguments to **popen** are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either 'r' for reading or 'w' for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by **popen** should be closed by **pclose**, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type 'r' command may be used as an input filter, and a type 'w' as an output filter.

## SEE ALSO

pipe(2), fopen(3), fclose(3), system(3), wait(2)

## DIAGNOSTICS

**popen** returns a null pointer if files or processes cannot be created, or the Shell cannot be accessed.
**pclose** returns $-1$ if *stream* is not associated with a 'popened' command.

## BUGS

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with **fflush**, see **fclose**(3).

## NAME

printf, fprintf, sprintf — formatted output conversion

## SYNOPSIS

**#include <stdio.h>**

**printf(format** [, arg ] ... **)**
**char *format;**

**fprintf(stream, format** [, arg ] ... **)**
**FILE *stream;**
**char *format;**

**sprintf(s, format** [, arg ] ... **)**
**char *s, format;**

## DESCRIPTION

**printf** places output on the standard output stream **stdout**. **fprintf** places output on the named output *stream*. **sprintf** places "output" in the string *s*, followed by the character '\0'.

Each of these functions converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg* **printf**.

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign '−' which specifies *left adjustment* of the converted value in the indicated field;
- an optional digit string specifying a *field width*; if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- an optional period '.' which serves to separate the field width from the next digit string;
- an optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e- and f-

conversion, or the maximum number of characters to be printed from a string;

- the character **l** specifying that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg*. (A capitalized conversion code accomplishes the same thing).

- a character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer *arg* supplies the field width or precision.

The conversion characters and their meanings are

**dox** The integer *arg* is converted to decimal, octal, or hexadecimal notation respectively.

**f** The float or double *arg* is converted to decimal notation in the style '[−]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

**e** The float or double *arg* is converted in the style '[−]d.ddde±dd' where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

**g** The float or double *arg* is printed in style **d**, in style **f**, or in style **e**, whichever gives full precision in minimum space.

**c** The character *arg* is printed. Null characters are ignored.

**s** *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

**u** The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 to 65535).

**%** Print a '%'; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by **printf** are printed by **putc**(3).

**Examples**

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour,
    min);
```

To print $\pi$ to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

**SEE ALSO**

putc(3), scanf(3), ecvt(3)

**BUGS**

Very wide fields (>128 characters) fail.

## NAME

putc, putchar, fputc, putw — put character or word on a stream

## SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
FILE *stream;
```

## DESCRIPTION

**putc** appends the character *c* to the named output *stream*. It returns the character written.

**putchar(***c***)** is defined as **putc(** *c*, **stdout)**.

**fputc** behaves like **putc**, but is a genuine function rather than a macro. It may be used to save on object text.

**putw** appends word (i.e. **int**) *w* to the output *stream*. It returns the word written. **putw** neither assumes nor causes special alignment in the file.

The standard stream **stdout** is normally buffered if and only if the output does not refer to a terminal; this default may be changed by **setbuf**(3). The standard stream **stderr** is by default unbuffered unconditionally, but use of **freopen** (see **fopen**(3)) will cause it to become buffered; **setbuf**, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. **fflush** (see **fclose**(3)) may be used to force the block out early.

**SEE ALSO**

fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

**DIAGNOSTICS**

These functions return the constant **EOF** upon error. Since this is a good integer, **ferror**(3) should be used to detect **putw** errors.

**BUGS**

Because it is implemented as a macro, **putc** treats a *stream* argument with side effects improperly. In particular 'putc(c, *f++);' doesn't work sensibly.

## NAME

puts, fputs — put a string on a stream

## SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

## DESCRIPTION

**puts** copies the null-terminated string *s* to the standard output stream **stdout** and appends a newline character.

**fputs** copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

## SEE ALSO

fopen(3), gets(3), putc(3), printf(3), ferror(3)
fread(3) for **fwrite**

## BUGS

**puts** appends a newline, **fputs** does not, all in the name of backward compatibility.

## NAME

qsort — quicker sort

## SYNOPSIS

**qsort(base, nel, width, compar)**
**char *base;**
**int (*compar)( );**

## DESCRIPTION

**qsort** is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

## SEE ALSO

sort(1)

## NAME

rand, srand — random number generator

## SYNOPSIS

**srand(seed)**
**int seed;**

**rand( )**

## DESCRIPTION

**rand** uses a multiplicative congruential random number generator with period $2^{32}$ to return successive pseudo-random numbers in the range from 0 to $2^{15} - 1$.

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. (The current time is not a bad choice).

## NAME

scanf, fscanf, sscanf — formatted input conversion

## SYNOPSIS

```
#include <stdio.h>

scanf(format [ , pointer ] . . . )
char *format;

fscanf(stream, format [ , pointer ] . . . )
FILE *stream;
char *format;

sscanf(s, format [ , pointer ] . . . )
char *s, *format;
```

## DESCRIPTION

**scanf** reads from the standard input stream **stdin**. **fscanf** reads from the named input *stream*. **sscanf** reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which match optional white space in the input.
2. An ordinary character (not %) which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

**%** a single '%' is expected in the input at this point; no assignment is done.

**d** a decimal integer is expected; the corresponding argument should be an integer pointer.

**o** an octal integer is expected; the corresponding argument should be a integer pointer.

**x** a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

**s** a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

**c** a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try '%1s'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

**e** a floating point number is expected; the next field is converted ac-
**f** cordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an 'E' or 'e' followed by an optionally signed integer.

**[** indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o**, and **x** may be capitalized or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters **e** or **f** may be capitalized or preceded by **l** to indicate a pointer to **double** rather than to **float**. The conversion characters **d**, **o**, and **x** may be preceded by **h** to indicate a pointer to **short** rather than to **int**.

The **scanf** functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done; if conversion was intended, it was frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf( "%d%f%s", &i, &x, name);
```

with the input line

```
25  54.32E-1  thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain 'thompson\0'. Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i,* 789.0 to *x,* skip '0123', and place the string '56\0' in *name*. The next call to **getchar** will return 'a'.

Since the newline character (\n) is used as a delimiter, it should not be matched as part of a literal string; if you use it in your format string, **scanf** will hang forever trying to match it from input. A **getchar()** can be used to swallow up an extra newline.

Note also that **scanf** only swallows up input it can match. This action can lead to unexpected behavior. For example, if **scanf** expects to read a number but is given a string, the string will remain on the input queue;

the next time **scanf** is called, it will immediately try to digest this string again, without waiting for another line to be entered.

**SEE ALSO**

atof(3), getc(3), printf(3)

**DIAGNOSTICS**

The **scanf** functions return **EOF** on end of input, and a short count for missing or illegal data items.

**BUGS**

The success of literal matches and suppressed assignments is not directly determinable.

## NAME

setbuf — assign buffering to a stream

## SYNOPSIS

**#include <stdio.h>**

**setbuf(stream, buf)**
**FILE *stream;**
**char *buf;**

## DESCRIPTION

**setbuf** is used after a stream has been opened but before it is read or written. It causes the character array *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer **NULL**, input/output will be completely unbuffered.

A manifest constant **BUFSIZ** tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from **malloc**(3) upon the first **getc** or **putc**(3) on the file, except that output streams directed to terminals, and the standard error stream **stderr** are normally not buffered.

## SEE ALSO

fopen(3), getc(3), putc(3), malloc(3)

## NAME

setjmp, longjmp — non-local goto

## SYNOPSIS

**#include <setjmp.h>**

**setjmp(env)**
**jmp_buf env;**

**longjmp(env, val)**
**jmp_buf env;**

## DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

**setjmp** saves its stack environment in *env* for later use by **longjmp**. It returns value 0.

**Longjmp** restores the environment saved by the last call of **setjmp**. It then returns in such a way that execution continues as if the call of **setjmp** had just returned the value *val* to the function that invoked **setjmp**, which must not itself have returned in the interim. All accessible data have values as of the time **longjmp** was called.

## SEE ALSO

signal(2)

## NAME

sin, cos, tan, asin, acos, atan, atan2 — trigonometric functions

## SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

## DESCRIPTION

**sin**, **cos**, and **tan** return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

**asin** returns the arc sin of $x$ in the range $-\pi/2$ to $\pi/2$.

**acos** returns the arc cosine of $x$ in the range 0 to $\pi$.

**atan** returns the arc tangent of $x$ in the range $-\pi/2$ to $\pi/2$.

**atan2** returns the arc tangent of $x/y$ in the range $-\pi$ to $\pi$.

## DIAGNOSTICS

Arguments of magnitude greater than 1 cause **asin** and **acos** to return value 0; **errno** is set to EDOM. The value of **tan** at its singular points is a huge number, and **errno** is set to ERANGE.

## BUGS

The value of **tan** for arguments greater than about 2**31 is garbage.

## NAME

sinh, cosh, tanh — hyperbolic functions

## SYNOPSIS

```
#include <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

**sleep(time)**

## DESCRIPTION

If *time* is greater than or equal to zero, the current process is suspended from execution for the number of seconds specified by the argument, up to 32767 seconds.

If *time* is less than zero, the current process is suspended for the number of clock-ticks (1/60ths of a second) equal in magnitude to *time*, up to 32768 ticks. The actual suspension time may be up to 1 clock-tick less than that requested, because scheduled wakeups occur at fixed 1-clock-tick intervals.

Because of scheduling delays due to other system activity, resumption of execution for sleep calls may be delayed an arbitrary amount.

The routine is implemented by setting an alarm clock signal (see **alarm**(2)) and pausing (**pause**(2)) until it occurs.

The following things will happen if a previous alarm was set in seconds:

> If the alarm was set to come due after the sleep would finish, then it will occur as scheduled after the sleep.
>
> If the previous alarm was set to come due during the sleep, then the sleep will terminate (prematurely) when the alarm occurs, and the alarm will be sent a second later.

Sleeps should *not* be mixed with clock-tick alarms.

If other signals are being caught, and one occurs during a sleep, then the signal-catching routine may itself be interrupted by the alarm ending the sleep; if this happens, then execution resumes normally at the point after the sleep call. In these circumstances the signal-catching routine will never get a chance to finish. To prevent this from happening, signal-catching routines can immediately call **signal** to ignore alarms.

## SEE ALSO

alarm(2), pause(2)

**NOTES**

Clock-tick sleeps are not portable to standard UNIX, and VENIX limits regular sleeps to 32767 seconds.

## NAME

stdio — standard buffered input/output package

## SYNOPSIS

**#include <stdio.h>**

**FILE *stdin;**
**FILE *stdout;**
**FILE *stderr;**

## DESCRIPTION

The functions described in Sections 3S constitute an efficient user-level buffering scheme. The in-line macros **getc** and **putc**(3) handle characters quickly. The higher level routines **gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf**, and **fwrite** all use **getc** and **putc**; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **FILE**. **fopen**(3) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. There are three normally open streams with constant pointers declared in the include file and associated with the standard open files:

| | |
|---|---|
| **stdin** | standard input file |
| **stdout** | standard output file |
| **stderr** | standard error file |

A constant 'pointer' **NULL** (0) designates no stream at all.

An integer constant **EOF** (−1) is returned upon end of file or error by integer functions that deal with streams.

Any routine that uses the standard input/output package must include the header file <**stdio.h**> of pertinent macro definitions. The functions and constants mentioned in sections labeled 3S are declared in the include file and need no further declaration. The constants, and the following 'functions' are implemented as macros; redeclaration of these names is perilous: **getc, getchar, putc, putchar, feof, ferror, fileno.**

## SEE ALSO

open(2), close(2), read(2), write(2)

## DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **FILE** pointer has not been initialized with **fopen**, input (output) has been attempted on an output (input) stream, or a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

## BUGS

Standard I/O is not usable in raw mode.

## NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex — string operations

## SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

## DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

**strcat** appends a copy of string *s2* to the end of string *s1*. **strncat** copies at most *n* characters. Both return a pointer to the null-terminated result.

**strcmp** compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. **strncmp** makes the same comparison but looks at *n* characters at most.

**strcpy** copies string *s2* to *s1*, stopping after the null character has been moved. **strncpy** copies exactly *n* characters, truncating or null-padding *s2*; the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

**strlen** returns the number of non-null characters in *s*.

**index (rindex)** returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

## BUGS

**strcmp** uses native character comparison, which is signed on PDP-11's, unsigned on other machines.

## NAME

swab — swap bytes

## SYNOPSIS

**swab(from, to, nbytes)**
**char *from, *to;**

## DESCRIPTION

**swab** copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11's and other machines. *nbytes* should be even.

## NAME

system — issue a shell command

## SYNOPSIS

**system(string)**
**char *string;**

## DESCRIPTION

**system** causes the *string* to be given to **sh**(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

## SEE ALSO

popen(3), exec(2), wait(2)

## DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

## NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs — terminal independent operation routines

## SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *tgetstr(id, area)
char *id, **area;

char *tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

## DESCRIPTION

These functions extract and use capabilities from the terminal capability data base **termcap**(5). These are low level routines; see **curses**(3) for a higher level package.

**tgetent** extracts the entry for terminal *name* into the buffer at *bp*. *bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to **tgetnum**, **tgetflag**, and **tgetstr**. **tgetent** returns −1 if it cannot open the **termcap** file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the **termcap** file.

If it does begin with a slash, the string is used as a path name rather than **/etc/termcap**. This can speed up entry into programs that call **tgetent**, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file **/etc/termcap**.

**tgetnum** gets the numeric value of capability *id*, returning −1 if is not given for the terminal. **tgetflag** returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. **tgetstr** gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in **termcap**(5), except for cursor addressing and padding information.

**tgoto** returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call **tgoto** should be sure to turn off the XTABS bit(s), since **tgoto** may now output a tab. Note that programs using **termcap** should in general turn off XTABS anyway since some terminals use control−I for other functions, such as nondestructive space). If a % sequence is given which is not understood, then **tgoto** returns "OOPS".

**tputs** decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable **ospeed** should contain the output speed of the terminal as encoded by **stty**(1). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (**^@**) is inappropriate.

The library switch for compilation is **−ltermlib**, and should be specified at the end of the **cc** command line.

## FILES

| | |
|---|---|
| /usr/lib/libtermlib.a | −ltermlib library |
| /etc/termcap | data base |

## SEE ALSO

ex(1), curses(3), termcap(5)

## NAME

ttyname, isatty, ttyslot — find name of a terminal

## SYNOPSIS

**char *ttyname(fildes)**

**isatty(fildes)**

**ttyslot()**

## DESCRIPTION

**ttyname** returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

**isatty** returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

**ttyslot** returns the number of the entry in the **ttys**(4) file for the control terminal of the current process.

Note that for every process **/dev/tty** is synonymous with the process' control terminal.

## FILES

/dev/*
/etc/ttys

## SEE ALSO

ioctl(2), ttys(4)

## DIAGNOSTICS

**ttyname** returns a null pointer (0) if *fildes* does not describe a terminal device in directory '/dev'.

**ttyslot** returns 0 if '/etc/ttys' is inaccessible or if it cannot determine the control terminal.

## BUGS

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

## NAME

ungetc — push character back into input stream

## SYNOPSIS

**#include <stdio.h>**

**ungetc(c, stream)**
**FILE *stream;**

## DESCRIPTION

**ungetc** pushes the character *c* back on an input stream. That character will be returned by the next **getc** call on that stream. **ungetc** returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push **EOF** are rejected.

**fseek**(3) erases all memory of pushed back characters.

## SEE ALSO

getc(3), setbuf(3), fseek(3)

## DIAGNOSTICS

**ungetc** returns **EOF** if it can't push a character back.

4

## NAME

a.out — assembler and link editor output

## SYNOPSIS

**#include <a.out.h>**

## DESCRIPTION

**a.out** is the output file of the assembler **as**(1) and the link editor **ld**(1). Both programs make **a.out** executable if there were no errors and no unresolved external references. Layout information as given in the include file is:

```
typedef long        AOUT_T;

/*
 * Header prepended to each a.out file.
 */
struct exec {
short               a_magic;    /* magic number */
unsigned short      a_stack;    /* size of stack if Z type, 0 otherwise */
long                a_text;     /* size of text segment */
long                a_data;     /* size of initialized data */
long                a_bss;      /* size of uninitialized data */
long                a_syms;     /* size of symbol table */
long                a_entry;    /* entry point */
long                a_trsize;   /* size of text relocation */
long                a_drsize;   /* size of data relocation */
};

#define  OMAGIC     0407    /* old impure format */
#define  NMAGIC     0411    /* read-only text (separate I&D) */

#define  SYMNMLEN 8         /* size of symbol name*/

/*
 * Macros which take exec structures as arguments and tell whether the
 * file has a reasonable magic number or offsets to text|symbols|strings.
 */
#define  N_BADMAG(x) \
         (long)(((x).a_magic)! = OMAGIC && ((x).a_magic)! = NMAGIC)
#define  N_TXTOFF(x) \
         (long)sizeof(struct exec)
```

```
#define N_SYMOFF(x) \
        (long)(N_TXTOFF(x) + (x).a_text+(x).a_data + \
        (x).a_trsize + (x).a_drsize)
#define  N_STROFF(x) \
        (long)(N_SYMOFF(x) + (x).a_syms)

/*
 * Format of a relocation datum.
 */
struct relocation_info {
long    r_address;          /* address which is relocated */
short   r_symbolnum;        /* local symbol ordinal */
short   r_pcrel:1,          /* was relocated pc relative already */
        r_length:2,         /* 0=byte, 1=word, 2=long */
        r_extern:1,         /* doesn't include value of sym referenced */
        :12;
};

/* Format of the old symbol table entry. This is here for compatibility.
 * The nlist subroutine takes an old symbol table format as its argument
 * and it knows how to read the format actually stored in the file.
 */
struct  nlist {
        char    n_name[SYMNMLEN];/* symbol name */
        short   n_type;  /* type */
        long    n_value; /* value */
};

/*
 * Format of a symbol table entry as it really is in the a.out file.
 */
struct  symtb {
        union {
                char    *ns_name;/* for use when in-core */
                unsigned short ns_strx;/* index into file string table */
        } ns_un;
        char    ns_type; /* type flag, i.e. N_TEXT etc; see below */
        char    ns_other;/* unused */
        short   ns_desc; /* see <stab.h> */
        long    ns_value;/* value of this symbol */
};

#define  ns_hash ns_desc  /* used internally by ld */
```

```
/*
 * Simple values for n_type or ns_type.
 */
#define  N_UNDF  0x0    /* undefined */
#define  N_ABS   0x2    /* absolute */
#define  N_TEXT  0x4    /* text */
#define  N_DATA  0x6    /* data */
#define  N_BSS   0x8    /* bss */
#define  N_COMM  0x12   /* common (internal to ld) */
#define  N_FN    0x1f   /* file name symbol */

#define  N_EXT   01     /* external bit, OR'ed in */
#define  N_TYPE  0x1e   /* mask for all the type bits */

#define  N_STAB  0xe0

/*
 * Format for namelist values.
 */
#define  N_FORMAT      "%08lx"
```

The file has four sections: a header, the program and data text, relocation information, a symbol table, and a string table (in that order). The last three may be empty if the program was loaded with the '-s' option of **ld** or if the symbols and relocation have been removed by **strip**(1).

In the header the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

## NAME

ar — archive (library) file format

## SYNOPSIS

**#include <ar.h>**

## DESCRIPTION

The archive command **ar** is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor **ld**(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 and header layout as described in the include file are:

```
#define  ARMAG      "!<arch>\n"
#define  SARMAG     8

#define  ARFMAG     "`\n"

struct ar_hdr {
        char    ar_name[16];
        char    ar_date[12];
        char    ar_uid[6];
        char    ar_gid[6];
        char    ar_mode[8];
        char    ar_size[10];
        char    ar_fmag[2];
};
```

## SEE ALSO

ar(1), ld(1), nm(1)

## NAME

checklist — default file system checklist file

## DESCRIPTION

**/etc/checklist** is used as a default checklist by a number of disk checking and reporting programs. The file is in the format:

```
filsys0:comment0:
filsys1:comment1:
filsys2:comment2:
    .
    .
    .
```

where filsys0, filsys1 ... are the names of the default devices, and comment0, comment1 ... are comment fields briefly summarizing their use. The two fields are separated by a colon, and a colon/newline occurs at the end of each entry.

## SEE ALSO

fsck(1), ncheck(1), df(1), quot(1)

## NAME

core — format of core image file

## DESCRIPTION

VENIX 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' working directory (provided it can be; normal access controls apply).

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

In general the debugger **adb**(1) is sufficient to deal with core images.

## SEE ALSO

adb(1), signal(2)

## NAME

dir — format of directories

## SYNOPSIS

**#include <sys/types.h>**
**#include <sys/dir.h>**

## DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry (see **filsys**(4)). The structure of a directory entry as given in the include file is:

```
#define DIRSIZ 14

struct direct{
        ino_t   d_ino;              /* inode number */
        char    d_name[DIRSIZ]; /* file name */
};
```

By convention, the first two entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory file system. Since there is no parent, '..' has the same meaning as '.'.

## SEE ALSO

filsys(4)

## NAME

filsys, flblk, ino — format of file system volume

## SYNOPSIS

**#include <sys/types.h>**
**#include <sys/flbk.h>**
**#include <sys/filsys.h>**
**#include <sys/ino.h>**

## DESCRIPTION

Every file system storage volume (e.g. RM disk, RK disk, RL disk, RX diskette), has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the 'super block'. The layout of the super block as defined by the include file **<sys/filsys.h>** is:

```
struct filsys {
        unsigned int    s_isize;        /* size in blocks of I list */
        unsigned int    s_fsize;        /* size of entire volume */
        int             s_nfree;        /* number of in-core free */
        unsigned int    s_free[100];    /* in-core free blocks */
        int             s_ninode;       /* number in-core I nodes */
        unsigned int    s_inode[100];   /* in core free I nodes */
        char            s_flock;        /* lock free list */
        char            s_ilock;        /* lock I list */
        char            s_fmod;         /* super block modified */
        char            s_ronly;        /* mounted read-only flag */
        long            s_time;         /* date of last update */
        int             pad[48];
};
```

*s_isize* is the number of blocks in the i-list, which starts just after the super-block, in block 2. *s_fsize* is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an 'impossible' block address is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free[1], ... , s_free[s_nfree−1],* up to 99 free block numbers. *s_free[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block number 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 numbers in the next 100 words into the *s_free* array. To free a block, check if *s_nfree* is 100; 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_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, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

*s_flock*, *s_ilock*, and *s_ronly* 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_time* is the last time the super-block of the file systems was changed, and is a double-precision representation of the number of seconds that have elapsed since 00:00:00 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.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 32 bytes long, so 16 of them fit into a block. Therefore, i-node *i* is located in block ( *i* + 31) / 16, and begins 32 * (( *i* +31) (mod 16)) from its start. I-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an inode is as follows:

```
struct inode {
        int             i_mode;
        char            i_nlinks;
        unsigned char   i_uid;
        unsigned char   i_gid;
        unsigned char   i_size0;
        unsigned int    i_size1;
        unsigned int    i_addr[8];
        long            i_atime;
        long            i_mtime;
};
```

The mode bits are as follows:

| | | |
|---|---|---|
| IALLOC | 0100000 | i-node is allocated |
| IFMT | 060000 | 2-bit file type mask: |
| IFDIR | 040000 | directory |
| IFCHR | 020000 | character type special file |
| IFREG | 000000 | regular file |
| IFBLK | 060000 | block-type special file |
| ILARG | 010000 | large file |
| ISUID | 04000 | set user-ID on execution |
| ISGID | 02000 | set group-ID on execution |
| ISVTX | 01000 | save shared segment after use |
| IREAD | 0400 | read (owner) |
| IWRITE | 0200 | write (owner) |
| IEXEC | 0100 | execute (owner) |
| | 0070 | read, write, execute (group) |
| | 0007 | read, write, execute (other) |

*i_mode* tells the kind of file; it is encoded identically to the *st_mode* field of **stat**(2). *I_nlink* is the number of directory entries (links) that refer to this i-node. *i_uid* and *i_gid* are the owner's user and group IDs. *i_size0* and *i_size1* are the 24-bit number of bytes in the file. *i_atime* and *i_mtime* are the times of last access and modification of the file contents (read, write or create). See **times**(2).

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the *i_addr* field is occupied by the device code (see **types**(5)). The device codes of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number *n* of a file is accessed as follows. *N* is divided by 512 to find its logical block number (say *b*) in the file. If the file is small (flag 010000 is 0), then *b* must be less than 8, and the physical block number is *addr[b]*.

If the file is large, *b* is divided by 256 to yield *i*. If *i* is less than 7, then *addr[i]* is the address of a first indirect block which contains the number of the block for the sought-for byte.

If *i* is equal to 7, then the file has become extra-large (huge), and *addr[7]* is the address of a first indirect block. Each word in this block is the number of a second-level indirect block; each word in the second-level indirect block points to a data block. Notice that extra-large files are not marked by any mode bit, but only by having *addr[7]* non-zero; and that although this scheme allows for more than 256x256x512 = 33,554,432 bytes per file, the length of files is stored in 24 bits so in practice a file can be at most 16,777,216 bytes long.

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

**SEE ALSO**

fsck(1), dir(4), mount(1), stat(2)

## NAME

group — group file

## DESCRIPTION

**group** contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory **/etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

## FILES

/etc/group

## SEE ALSO

newgrp(1), crypt(3), passwd(1), passwd(4)

## NAME

mtab — mounted file system table

## DESCRIPTION

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

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

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

## FILES

/etc/mtab

## SEE ALSO

mount(1)

## NAME

passwd — password file

## DESCRIPTION

**passwd** contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID (0 – 255)
numerical group ID (0 – 255)
GCOS job number, box number, optional GCOS user-id
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /**etc**. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

## FILES

/etc/passwd

## SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(4)

## NAME

ttys — terminal initialization data

## DESCRIPTION

The **ttys** file is read by the **init** program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to **getty**, (see section 8, *Installation and System Manager's Guide*) which performs such tasks as baud-rate recognition, reading the login name, and calling **login**. The remainder of the line is the terminal's entry in the device directory, **/dev**.

For normal lines, the second character on the line (passed to **getty**) is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary. The following is a complete list:

**0** Cycles through 300 – 1200 – 150 – 110 baud. Useful as a default for dialup lines accessed by a variety of terminals. (An interface with software-controllable baud rates is required for this to be effective).

– Intended for the console terminal.

**1** Intended for on-line CRT terminals (9600 baud).

**2** Intended for on-line VT52 and VT100 terminals (9600 baud). Like 1 (above), but screen is cleared with ESC – H ESC – J sequence when login prompt is issued.

**3** Starts at 1200 baud, cycles to 300 baud and back. Useful with 212 datasets where most terminals run at 1200 baud.

**4** Useful for on-line DECwriter(LA36).

**5** Same as '3' but starts at 300.

**6** Cycles through 300 – 1200 – 9600 baud, starting at 300 baud.

**7** Same as '6', but starts at 1200.

**8** Same as '6', but starts at 9600.

**A** 2400 baud line.

**B** 4800 baud line.

Only as many user ports as licensed will become active; attempts to set additional ports active will be ignored. It is entirely permissible to specify less than the licensed number of active ports.

**FILES**

/etc/ttys

**SEE ALSO**

login(1), getty (8)

## NAME

utmp, wtmp — login records

## SYNOPSIS

**#include <utmp.h>**

## DESCRIPTION

The **utmp** file allows one to discover information about who is currently using VENIX. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
        char ut_line[8];        /* tty name */
        char ut_name[8];        /* user id */
        long ut_time;           /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of **time**(2).

The **wtmp** file records all logins and logouts. Its format is exactly like **utmp** except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names '|' and '}' indicate the system-maintained time just before and just after a **date** command has changed the system's idea of the time.

**wtmp** is maintained by **login**(1) and **init** (section 8, *Installation and System Manager's Guide*). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by **ac**(1).

## FILES

/etc/utmp
/usr/adm/wtmp

## SEE ALSO

login(1), init(8), who(1), ac(1)

5

## 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 are 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**(4).

**TERM** The kind of terminal for which output is to be prepared. Some terminals supported are:

| | |
|---|---|
| vi200 | Visual Technologies |
| vi50 | |
| vt52 | DEC |
| vt100 | |
| h19 | Zenith |

See a complete listing of terminals in **/etc/termcap.**

**MAIL** If this variable is set to the name of a mail file, then the shell informs the user of the arrival of mail in the specified file.

Further names may be placed in the environment by the **export** command and "name=value" arguments in **sh**(1).

It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL, PS1, PS2, IFS.**

## SEE ALSO

login(1), sh(1), getenv(3C)

## NAME

plot — graphics interface

## DESCRIPTION

Files of this format are produced by routines described in **plot**(3g), and are interpreted for various devices by **plot**(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.

There are no instructions for **openpl**( ) or **closepl**( ), as they are automatically invoked by the filter driver when the **plot**(1g) command is given.

Each of the following descriptions begins with the name of the corresponding routine in **plot**(3g). See **plot**(3g) for a more detailed description of these routines.

Standard Unix routines

**e** erase: Erase the graphics screen. No bytes follow.

**s** space: The next two coordinate pairs (four bytes each) give the lower-left and upper-right corners of the user-coordinate system to be defined.

**t** label: Place the following ASCII string, ending with a newline, so that its first character falls on the current point.

**l** line: The next two coordinate pairs (four bytes each) specify the start and end points of the line to be drawn.

**c** circle: The next coordinate pair (four bytes) specify the center and the following two bytes specify the radius of a complete circle.

**a** arc: The next three coordinate pairs (four bytes each) give the center point, and starting and ending octants of a counterclockwise circular arc.

**m** move: The next four bytes give a new current point.

**n** cont: Draw a line from the old current point to the new current point given by the next four bytes.

**p** point: Plot a point at the new current point, given by the next four bytes.

**f** linemod: The following ASCII string, ending with a newline, chooses one of the line-styles available in the graphics filter.

Special VENIX enhancements

**g** linepat: The next two bytes specify a 16-bit integer pattern, of the user's choice, to be used as a special line pattern.

**h** linewid: The next two bytes set desired width, in user coordinates, of all lines following.

**w** window: The next two coordinate pairs (four bytes each) specify bottom-left and upper-right corners of a clipping window in user coordinates.

**b** box: The next two coordinate pairs (four bytes each) specify the bottom-left and upper-right corners of a rectangular box.

**r** rfill: The next two coordinate pairs (four bytes each) specify the lower-left and upper-right corners of a filled rectangular box.

**u** fill: The next four bytes specify the coordinates of a seed point for a general convex boundary fill.

**d** dot: The next coordinate pair (four bytes) specify the center location and the following two bytes specify the radius of a filled circle.

**o** color: The next two bytes specify a choice from the color palette. All graphics following will be drawn in the specified color.

**i** colndx: The first two bytes specify the palette color to be modified, and the next two bytes specify the new color.

**j** writemod: The following ASCII string, ending with a newline, chooses one of the writing modes available in the graphics filter.

**SEE ALSO**

plot(1g), plot(3g)

## NAME

termcap — terminal capability data base

## SYNOPSIS

**/etc/termcap**

## DESCRIPTION

**termcap** is a data base describing terminals. It is used by **vi**(1) and **curses**(3) and is accessible by user programs. Terminals are described in **/etc/termcap** which gives a set of terminal capabilities, and how operations are performed. Padding requirements and initialization sequences are included in **termcap**. We have provided definitions of a dozen popular terminals; you may want to add your own.

Entries in **termcap** consist of a number of ':' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name is always 2 characters long and is only used by older version 6 UNIX systems (not applicable to the Professional 350). The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

## CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on 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**(1) 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 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, 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:al=3*\E^R:am:bs:cd=16*\E^C:\
        :ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :co#80:\
        :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 if '\' is given as the last character of a line, and empty fields (i.e. extra colons) may be included for readability (here between the last field on a line and the first field on the next). Capabilities in **termcap** are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

**Types of Capabilities**

All capabilities have two letter codes. For instance, the fact that the Concept has "automatic margins" (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value '80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an '=', and then a string ending at the next following ':'. A delay in milliseconds may appear after the '=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either an 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' to specify a delay per unit in tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A **\E** maps to an ESCAPE character, **^x** maps to a control-x for any appropriate *x*, and the

sequences **\n \r \t \b \f** give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a ****, and the characters ^ and **** may be given as **\^** and ****. If it is necessary to place a **:** in a capability it must be escaped in octal as **\072**. If it is necessary to place a null character in a string capability it must be encoded as **\200**. The routines which deal with **termcap** use C strings, and strip the high bits of the output very late so that a **\200** comes out as a **\000** would.

**Preparing Descriptions**

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in **termcap** and to build up a description gradually, using partial descriptions with **ex** to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **termcap** file to describe it or bugs in **ex**. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in **/etc/termcap**. TERMCAP can also be set to the **termcap** entry itself to avoid reading the file when starting up the editor.

**Basic capabilities**

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than **^H** (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in **termcap** are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that linefeeding from 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 the 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**

**termcap** entries can describe two basic different mechanisms used by intelligent terminals to insert/delete characters. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept-100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (or an empty value if **im** is empty). 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**(1), 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), termcap(3), vi(1)

## 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

types — system type declarations

## SYNOPSIS

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

## DESCRIPTION

Various system calls (e.g. **stat**(2)) return system information in variables of specific types. For portability purposes, these types are all declared in the include file <**sys/types.h**> and are given below:

```
typedef  unsigned int  daddr_t;  /* disk address */
typedef  char *        caddr_t;  /* core address */
typedef  int           ino_t;    /* i-node number */
typedef  long          time_t;   /* a time */
typedef  int           dev_t;    /* device code */
```

## SEE ALSO

stat(2)

6

## NAME

backgammon — the game

## SYNOPSIS

**/usr/games/backgammon**

## DESCRIPTION

This program does what you expect. It will ask whether you need instructions.

## NAME

banner — make long posters

## SYNOPSIS

**/usr/games/banner**

## DESCRIPTION

**banner** reads the standard input and prints it sideways in huge built-up letters on the standard output.

## 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 (^C) and the action and standing will be printed.

## NAME

checkers – game

## SYNOPSIS

**/usr/games/checkers**

## DESCRIPTION

**checkers** uses standard notation for the board:

BLACK

| //// //// | 1 | //// //// | 2 | //// //// | 3 | //// //// | 4 |
|---|---|---|---|---|---|---|---|
| 5 | //// //// | 6 | //// //// | 7 | //// //// | 8 | //// //// |
| //// //// | 9 | //// //// | 10 | //// //// | 11 | //// //// | 12 |
| 13 | //// //// | 14 | //// //// | 15 | //// //// | 16 | //// //// |
| //// //// | 17 | //// //// | 18 | //// //// | 19 | //// //// | 20 |
| 21 | //// //// | 22 | //// //// | 23 | //// //// | 24 | //// //// |
| //// //// | 25 | //// //// | 26 | //// //// | 27 | //// //// | 28 |
| 29 | //// //// | 30 | //// //// | 31 | //// //// | 32 | //// //// |

WHITE

Black plays first. The program normally plays white. To specify a move, name the square moved from and the square moved to. For multiple jumps name all the squares touched.

Certain commands may be given instead of moves:

reverse Reverse roles; the program takes over your pieces.

backup Undo the last move for each player.

list Print the record of the game.

move Let the program select a move for you.

print Print a map of the present position.

## NAME

chess — the game of chess

## SYNOPSIS

**/usr/games/chess**

## DESCRIPTION

**chess** is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '+' is used to specify check; 'o-o' and 'o-o-o' specify castling. To play black, type 'first'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

Type 'exit' to stop the game.

## FILES

/usr/lib/book opening 'book'

## DIAGNOSTICS

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

## WARNING

Over-use of this program will cause it to go away.

## BUGS

Pawns may be promoted only to queens.

## NAME

fortune — fortune cookie

## SYNOPSIS

**/usr/games/fortune**

## DESCRIPTION

**fortune** prints a one-line fortune of inestimable value. It is commonly executed on login from a user's *.profile*.

## FILES

| | |
|---|---|
| /usr/games/lib/fortunes | fortune library |

## NAME

maze — generate a maze problem

## SYNOPSIS

**/usr/games/maze/**

## DESCRIPTION

**maze** asks a few questions and then prints a maze.

## BUGS

Some mazes (especially small ones) have no solutions.

Floating point hardware is required.

## 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

quiz — test your knowledge

## SYNOPSIS

**/usr/games/quiz** [ **−i** file ] [ **−t** ] [ category1 category2 ]

## DESCRIPTION

**quiz** gives associative knowledge tests on various subjects. It asks items chosen from *category1* and expects answers from *category2*. If no categories are specified, **quiz** gives instructions and lists the available categories.

**quiz** tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, **quiz** reports a score and terminates.

The **−t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The **−i** flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

```
line      = category newline | category ':' line
category  = alternate | category '|' alternate
alternate = empty | alternate primary
primary   = character | '[' category ']' | option
option    = '{' category '}'
```

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash '\' is used as with **sh**(1) to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, **quiz** will refrain from asking it.

## FILES

/usr/games/quiz.k/*

## BUGS

The construct 'a|ab' doesn't work in an information file. Use 'a{b}'.

## NAME

ttt, cubic — tic-tac-toe

## SYNOPSIS

**/usr/games/ttt**

**/usr/games/cubic**

## 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.

**cubic** plays three-dimensional tic-tac-toe on a 4×4×4 board. Moves are specified as a sequence of three coordinate numbers in the range 1 – 4.

## FILES

| | |
|---|---|
| /usr/games/ttt.k | learning file |

## NAME

wump — the game of hunt-the-wumpus

## SYNOPSIS

**/usr/games/wump**

## DESCRIPTION

**wump** plays the game of 'Hunt the Wumpus.' A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company*, *2*, 2 (November 1973).

## BUGS

It will never replace Space War.

Printed in U.S.A.

AA-CM95A-XH