DOMAIN/IX Programmer's Reference for BSD4.2

Order No. 005801
Revision 01

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

Printed in U.S.A.
First Printing: December 1986

This document was formatted on a DOMAIN System using the **troff** text formatter distributed with DOMAIN®/IX software.

PREFACE

The DOMAIN®/IX™ *Programmer's Reference Manual for BSD4.2* consists of material on system calls, library functions, special (e.g., device) files, and other information of interest to programmers developing applications that run on DOMAIN/IX or other implementations of the UNIX® Operating System.

Audience

This *Programmer's Reference Manual* is intended for system and applications programmers and other knowledgeable users who are familiar with BSD4.2 UNIX software and DOMAIN networks. We recommend that you read one of the following tutorial introductions if you are not already familiar with the UNIX operating system.

- Bourne, Stephen W. *The UNIX System.* Reading: Addison-Wesley, 1982.
- Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment*, Englewood Cliffs, Prentice-Hall, 1984.
- Thomas, Rebecca and Jean Yates. *A User Guide to the UNIX System.* Berkeley: Osborne/McGraw-Hill, 1982.

This document also assumes a basic familiarity with the DOMAIN/IX system. The best introduction to the DOMAIN/IX system is *Getting Started With Your DOMAIN/IX System* (Order No. 008017). This manual explains how to use the keyboard and display, read and edit text, and manipulate files. It also shows how to request DOMAIN system services using interactive commands.

The Structure of This Document

This manual includes the following sections.

Section 2	provides reference material on system calls.
Section 3	provides reference material on library functions.
Section 4	provides reference material on devices and other "special" files.
Section 5	provides reference material on file formats.
Section 7	is a collection of miscellaneous information.

UNIX is a Registered Trademark of AT&T.

Sections 1 (user commands) and 6 (games) are in the *DOMAIN/IX Command Reference Manual.* Section 8 (administrative commands) is in the *DOMAIN/IX Administrator's Reference Manual.*

Related Volumes

The *DOMAIN/IX User's Guide* (Order No. 005803, revision 01) is the first volume you should read. It explains how DOMAIN/IX works, and contains extensive material on the Bourne Shell, C Shell, and **Mail**.

The *DOMAIN/IX Text Processing Guide* (Order No. 005803) describes the UNIX text editors (**ed**, **ex**, and **vi**) supported by DOMAIN/IX. It also contains material on the formatters **troff** and **nroff**, the macro packages **ms**, **me**, and **mm**, and the preprocessors **eqn** and **tbl**.

The *DOMAIN/IX Support Tools Guide* (Order No. 009413) describes various DOMAIN/IX utilities (e.g.**awk**(1), **lex**(1), **yacc**(1), etc.) that can help with development and maintenance of programs.

The *DOMAIN/IX Command Reference for System V* (Order No. 005798, revision 01) describes all the UNIX System V shell commands supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Programmer's Reference for System V* (Order No. 005799, revision 01) describes all the UNIX System V system calls and library functions supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Administrator's Reference for System V* (Order No. 009356) describes all the UNIX System V system administrator commands and provides detailed information on system registries and servers supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN/IX Command Reference for BSD4.2* (Order No. 005800, revision 01) describes all the BSD4.2 UNIX shell commands supported by the *bsd4.2* version of DOMAIN/IX.

The *DOMAIN/IX Administrator's Reference for BSD4.2* (Order No. 009355) describes all the UNIX System V system administrator commands and provides detailed information on system registries and servers supported by the *sys5* version of DOMAIN/IX.

The *DOMAIN C Language Reference* (Order No. 002093) describes C program development on the DOMAIN system. It lists the features of C, describes the C library, and gives information about compiling, binding, and executing C programs.

The *DOMAIN System Command Reference* (Order No. 002547) gives information about using the DOMAIN system and describes the DOMAIN commands found in the */com* directory.

The two-volume *DOMAIN System Call Reference* (Volume I Order No. 007196 revision 01, Volume II Order No. 007194 revision 01) describes calls to operating system components that are accessible to user programs.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

bold We use **bold** type to emphasize keywords in text and command-line examples. A keyword can be any of:

- The name of an executable system object (command or shell script) and any options (switches, regular expressions, or real pathnames) that command or shell script accepts. For example, **ls -la**, or **man ls**.
- The name of a callable function, including all syntactically required punctuation. For example, **open** (*path*, *flags*, *mode*).
- Any system object that has its own reference manual entry. For example, **passwd**(4).

We do not use bold type for general emphasis. In our ASCII help files, bold type looks the same as Roman type.

Italics We use *Italics* to emphasize:

- Names or pathnames of system objects. For example */etc/passwd* or */tmp*.
- Names we use as stand-ins for names and/or values that you must supply. For example, **man** *foo*, "...prints *filename* on standard output...," **open** (*path*, *flags*, *mode*). An example command line like

  ls *[options] [file(s)]*

  indicates that **ls** is a keyword that can be followed with one or more *options* and an optional *file* or *files*.

By extension, this font usage appears in command options and option arguments:

-n *number*	Number of times to do this function

as well as in function arguments

#include <sys/file.h>

open(*path, flags, mode*)
char **path*;
int *flags, mode*;

We also use italics to indicate the title of a publication, such as the

DOMAIN/IX Command Reference Manual. We do not use Italic type for general emphasis. In our ASCII help files, Italic type is underlined.

pica — Where possible, we use the constant-width pica font (or another "type-writer" style font) in code fragments, shell or DM scripts, and scripts for commands like **awk**(1) and **sed**(1). In our ASCII help files, pica type looks the same as Roman type.

name(1) — Where a filename or command name is followed by a number or number-letter pair in parentheses, that number indicates the section (and, if a letter is included, the subsection) of the reference manual set in which you can find reference information on the named command or file. For example, you can find reference information on the **lex**(1) command in Section 1 of the *DOMAIN/IX Command Reference Manual* and information on the */etc/passwd*(4) file in Section 4 of the *DOMAIN/IX Programmer's Reference Manual.*

[brackets] — We use brackets to delimit optional command line switches (options) and arguments. Brackets are also shell metacharacters that delimit a range or character class.

<KEY> — We enclose the name of a keyboard key in brackets. For example, <ESC> or <AGAIN>. The < and > symbols are also shell metacharacters used for redirection of input or output.

↑<KEY> — A control function that you execute by pressing the <CTRL> key and the named <KEY> at the same time. For example, ↑<D> sends an End-Of-File.

<CTRL><KEY> Same as ↑<KEY>.

... — Horizontal ellipses indicate that the preceding item can be repeated an arbitrary number of times. For example

troff *file* ...

means that you can say

troff *file1 file2 file3*

and so on.

.
.
. — We use vertical ellipses to indicate that an irrelevant portion of text has been omitted from an example.

Note that, when we begin a sentence with the name of a filesystem object, we always capitalize the first letter of the name unless this would result in an ambiguity.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels, you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the CRUCR (Create User Change Request) command. You can also get more information by typing:

/com/help crucr

in any UNIX or AEGIS shell. There is a Reader's Response form at the back of this manual. We'd appreciate it if you would take the time to fill it out when you're ready to comment on this document.

NAME

intro – introduction to system calls and error numbers

USAGE

#include <errno.h>

DESCRIPTION

In this section of the *Programmer's Reference Manual*, we describe all of the UNIX system calls available under the *bsd4.2* version of DOMAIN/IX. Typically, these calls return zero or some positive integer when they succeed, and -1 (or another "impossible" return value) if they fail. Details are provided in the individual descriptions.

As with normal arguments, all return codes and values from functions are of type **int** (integer) unless otherwise noted. In addition, an error number is also made available in the external variable *errno*. Since *errno* is not cleared on successful calls, it should be tested only after an error has occurred.

In this introduction, we list the various values and meanings for *errno*, and also provide a glossary of the terms we use in this section and subsequent sections of this manual.

ERROR NUMBERS

The following is a complete list of the errors and their names as given in *<errno.h>*.

Kernel Errors

0 unused

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way that is forbidden to anyone but the file's owner or the super-user. It also may indicate an attempt by an ordinary user to do something permitted only to the super-user.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process

The process whose number was given to **kill**(2) 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**(2) or **write**(2). Occasionally, this error occurs on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or attempts to read/write beyond the limits of the device. It may also occur when, for example, an illegal tape drive unit number is selected.

7 E2BIG Arg list too long
An argument list longer than 10240 bytes is presented to **execve**(2).

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, is not of the correct type.

9 EBADF Bad file number
A file descriptor refers to no open file, or a read (write) request is made to a file which is open only for writing (reading).

10 ECHILD No children
A **wait** was executed by a process with no living or unwaited-for children.

11 EAGAIN No more processes
A **fork**(2) was attempted when the system's process table was full.

12 ENOMEM Not enough memory
During an **exec**(2), **break**(2), or **sbrk**(2), a program asks for more memory than the system is able to supply.

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 non-block file was mentioned where a block device was required.

16 EBUSY Device busy
An attempt was made to acquire a device that is already acquired or an release a device on which there is an active file directory.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g. **link**(2).

18 EXDEV Cross-device link
An attempt was made to create a hard link to a file on another device.

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
Something that is not a directory was specified where a directory is required, for example in a path name or as an argument to **chdir**(2).

21 EISDIR Is a directory
An attempt was made to 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. No more **opens** can succeed unless a currently-open file is first closed.

24 EMFILE Too many open files
A process has exceeded the DOMAIN System limit of 128 open file descriptors.

25 ENOTTY Not a character device
The file mentioned in an **ioctl**(2) is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt was made to execute a shell script that is currently open for writing, or to write to a shell script that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size set by **ulimit**(2).

28 ENOSPC No space left on device
A **write** was attempted to an ordinary file when there was no free space left on the device.

29 ESPIPE Illegal seek
An **lseek** was issued to a pipe.

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

31 EMLINK Too many links
An attempt was made to establish more than 1000 hard links to a file.

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

Math Library Errors

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.

Interprocess Communication Errors

35 EWOULDBLOCK Operation would block
An operation that would cause a process to block was attempted on a object in non-blocking mode (see **ioctl**).

36 EINPROGRESS Operation now in progress
An operation that takes a long time to complete (such as a **connect**(2)) was attempted on a non-blocking object (see **ioctl**).

37 EALREADY Operation already in progress
An operation was attempted on a non-blocking object which already had an operation in progress.

38 ENOTSOCK Socket operation on non-socket
A socket operation was attempted on something that is not a socket.

39 EDESTADDRREQ Destination address required
A required address was omitted from an operation on a socket.

40 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer.

41 EPROTOTYPE Protocol wrong type for socket
A protocol was specified which does not support the semantics of the socket type requested. For example you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 ENOPROTOOPT Bad protocol option
A bad option was specified in a **getsockopt**(2) or **setsockopt**(2) call.

43 EPROTONOSUPPORT Protocol not supported
The requested protocol is not supported on the system.

44 ESOCKTNOSUPPORT Socket type not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

45 EOPNOTSUPP Operation not supported on socket
An operation was attempted on a socket type that does not support it (e.g., trying to **accept**(2) a connection on a datagram socket.)

46 EPFNOSUPPORT Protocol family not supported
The protocol family has not been configured into the system or no implementation for it exists.

47 EAFNOSUPPORT Address family not supported by protocol family
Th specified address was incompatible with the requested protocol. For example, you shouldn't necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

48 EADDRINUSE Address already in use
Only one usage of each address is normally permitted.

49 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine.

50 ENETDOWN Network is down
A socket operation encountered a dead network.

51 ENETUNREACH Network is unreachable
A socket operation attempted to reach a socket on an unreachable network.

52 ENETRESET Network dropped connection on reset
The host you were connected to crashed and rebooted.

53 ECONNABORTED Software caused connection abort
A connection abort was caused by your host machine.

54 ECONNRESET Connection reset by peer
A connection was forcibly closed by a peer. This normally results from the peer executing a **shutdown**(2) call.

55 ENOBUFS No buffer space available
An operation on a socket or pipe failed because the system lacked sufficient buffer space.

56 EISCONN Socket is already connected
A **connect**(2) was requested to a socket that is already connected, or a **sendto**(2) or **sendmsg**(2) request on a connected socket specified a destination other than the connected party.

57 ENOTCONN Socket is not connected
An request to send or receive data failed because the specified socket is not connected.

58 ESHUTDOWN Can't send after socket shutdown
A request to send data failed because the socket had already been shut down (see **shutdown**(2)).

59 unused

60 ETIMEDOUT Connection timed out
A **connect** request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

61 ECONNREFUSED Connection refused
No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service which is inactive on the foreign host.

62-74 unused

75 EHOSTUNREACH Host is unreachable
An attempt was made to reach an unreachable host.

76 ENOTEMPTY Directory not empty
An attempt was made to remove a directory that is not empty.

DEFINITIONS

Process ID — Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30,000.

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

Process Group ID — Each active process is a member of a process group that is identified by a positive integer called the process group ID. This is the process ID of the group leader. This grouping permits the signalling of related processes (see **killpg**(2)) and the job control mechanisms of **csh**(1).

Tty Group ID — Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to arbitrate between multiple jobs contending for the same terminal; see **csh**(1), and **tty**(4).

User ID and **Group ID** — Each user on the system is identified by a positive integer termed the user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

All processes have a user ID and group ID. These are initialized from the equivalent attributes of the process which created it.

Effective User Id, Effective Group Id, and **Access Groups** — Access to system resources is governed by three values: the effective user ID, the effective group ID, and the group access list.

The effective user ID and effective group ID are initially the process's real user ID and real group ID respectively. Either may be modified through execution of a set-user-ID or set-group-ID file (possibly by one its ancestors); see **execve**(2).

The group access list is an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in "File Access Permissions".

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

Special Processes — On DOMAIN systems, the processes with process ID's 1-11 are considered "special." Process 1 is normally Display Manager (DM) on DOMAIN nodes and the Server Process Manager (SPM) on DOMAIN Server Processors. It is the ancestor of every other process in the system. It is used to control the process structure. Other special processes include the Null Process (usually process 2), the Clock, the Page Purifier, and the network service processes.

Descriptor — This is an integer assigned by the system when a file is referenced by **open**(2), **dup**(2), or **pipe**(2) or a socket is referenced by **socket**(2) or **socketpair**(2) which uniquely identifies an access path to that file or socket from a given process or any of its children.

Filename — Names consisting of up to 32 characters may be used to name an ordinary file, special file, or directory.

These characters may be selected from the set of all ASCII characters excluding 0 (null) and 47 (slash).

Note that it is generally unwise to use *, ?, [or] in filenames. These characters have special meaning to the shell.

Pathname — A pathname is a null-terminated character string that includes zero or more directory names separated by slashes, optionally followed by a file name. The total length of a path name must be less than {PATHNAME_MAX} characters.

If a path name begins with a slash, the path search begins at the node's entry (root) directory. If a path name begins with a double slash, the path search begins at the network root, a list of all nodes on the network. Otherwise, the search begins from the current working directory. A slash by itself names the node's entry directory. A null pathname refers to the current directory.

Directory — A directory is a special type of file which contains entries that are references to other files. Directory entries are referred to as links. By convention, each directory contains at least two links, "." and "..", referred to as "dot" and "dot-dot" respectively. Dot is a link to the directory itself and dot-dot is a link to its parent directory. DOMAIN/IX does not currently observe this convention.

Root Directory and **Current Working Directory** — Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process's root directory need not be the node's root directory.

File Access Permissions — Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be changed at some later time through the **chmod**(2) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

- The process's effective user ID is that of the super-user.
- The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.
- The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the group permissions allow the access.
- Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the permissions for "other users" allow access.

Otherwise, permission is denied.

Note: DOMAIN/IX also supports Access Control Lists (ACLs), a different, finer-grained protection mechanism. ACLs and their interaction with the standard UNIX protection mechanism are described in detail in the *DOMAIN/IX Administrator's Reference Manual.*

Sockets and Address Families — A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult **socket**(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

RELATED INFORMATION

intro(3), **perror**(3)

NAME

accept – accept a connection on a socket

USAGE

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

ns = accept(*s*, *addr*, *addrlen*)
int *ns*, *s*;
struct sockaddr **addr*;
int **addrlen*;

DESCRIPTION

Accept takes the first connection from the queue of connections waiting at a socket *s*, creates a new socket with the properties of the original one, and allocates a file descriptor, *ns*, for the new socket. The original socket *s* was created with **socket** (2) and was bound to an address with **bind**(2). *S* is now listening for connections after a **listen**(2).

If there are no connections waiting and the socket is not marked as non-blocking, **accept** blocks the caller until a connection is present. If the socket is marked as non-blocking and no connections are waiting, **accept** returns an error (see below). The new accepted socket, *ns*, may not accept more connections. The original socket *s*, however, remains open.

The argument *addr* is a result parameter, which is filled in with the address of the connecting entity. The environment in which communications take place determines the exact format of the *addr* parameter. *Addrlen* is a value-result parameter; it should initially contain the amount of space that *addr* points to; upon return, it contains the actual length (in bytes) of the address returned. You can use this call with connection-based socket types, currently with SOCK_STREAM.

You may **select**(2) a socket for the purposes of doing an **accept** by selecting it for read.

RETURN VALUE

A successful **accept** returns a non-negative integer, which is the descriptor for the accepted socket. Otherwise, **accept** returns -1 and sets *errno* as indicated below.

ERRORS

The **accept** will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor refers to a file, not a socket.

[EOPNOTSUPP]	The socket is not of the type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked as non-blocking and no connections are waiting.

RELATED INFORMATION

bind(2), connect(2), listen(2), select(2), socket(2)

NAME

access – determine if a file can be accessed

USAGE

```
#include &lt;sys/file.h&gt;

#define R_OK    4    /* test for read permission */
#define W_OK    2    /* test for write permission */
#define X_OK    1    /* test for execute (search) permission */
#define F_OK    0    /* test for presence of file */

accessible = access(path, mode)
int accessible;
char *path;
int mode;
```

DESCRIPTION

Access checks the given file *path* for access rights according to *mode*, which is an inclusive OR of the bits R_OK, W_OK, and X_OK. Specifying *mode* as F_OK (i.e., zero) tests whether the directories leading to the file can be searched and whether the file exists.

Access uses the real user ID and the group access list (including the real group ID) to verify permission, making it useful in set-UID programs.

Note that **access** only checks access bits. A directory may appear writable according to **access**, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but **execve**(2) will fail unless the file is in the proper format.

RETURN VALUE

A successful **access** returns zero. If *path* cannot be found, or if any of the desired access modes would not be granted, **access** returns -1 and sets *errno* as indicated below.

ERRORS

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

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

[ENOENT] The argument pathname was too long.

[ENOENT] Read, write, or execute (search) permission is requested for a null pathname, or the named file does not exist.

[EPERM] The argument contains a byte with the high-order bit set.

[ELOOP] The call encountered too many symbolic links in translating the pathname.

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

[EACCES] Permission bits of the file mode do not permit the requested access; or search permission is denied on a component of the path prefix. The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group (other than the owner) have permission checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

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

RELATED INFORMATION

chmod(2), stat(2)

NAME

bind – bind a name to a socket

USAGE

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

bind(*s*, *name*, *namelen***)**
int *s*;
struct sockaddr **name*;
int *namelen*;

DESCRIPTION

Bind assigns a *name* to an unnamed socket. When a socket is created with **socket(2)**, it exists in a name space (address family) but has no *name* assigned. **Bind** requests that *name* be assigned to the socket. The rules used in name binding vary among communications environments.

RETURN VALUE

A successful **bind** returns zero. Otherwise, **bind** returns -1 and sets *errno* as indicated below.

ERRORS

Bind will fail if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The **name** parameter is not in a valid part of the user address space.

RELATED INFORMATION

connect(2), listen(2), socket(2), getsockname(2)

NAME

brk, sbrk – change data segment size

USAGE

caddr_t brk(*addr***)**
caddr_t *addr***;**

caddr_t sbrk(*incr***)**
int *incr***;**

DESCRIPTION

The system's idea of the lowest data segment location not used by the program is called the break. **Brk** sets the break to *addr* (rounded up to the next multiple of the system's page size). Locations greater than *addr* and below the stack pointer are not in the address space and will therefore cause a memory violation if the program attempts to access them.

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

When a program begins execution with an execve(2), the break is set at the highest location defined by the program and data storage areas. Consequently, programs that grow their data area are the principal clients of **sbrk**.

RETURN VALUE

A successful call to *brk* or **sbrk** returns zero and sets or extends the break. Otherwise, it returns -1 and sets *errno* as indicated below.

ERRORS

Sbrk will fail if one of the following is true:

[ENOMEM] The system's memory limit was exceeded.

[ENOMEM] The maximum possible size of a data segment (compiled into the system) was exceeded.

RELATED INFORMATION

execve(2), malloc(3)

NAME

chdir – change current working directory

USAGE

chdir(*path*)
char **path*;

DESCRIPTION

Chdir sets *path*, which must be the name of a directory, as the current working directory. This becomes the starting point for resolving pathnames not beginning with a slash (/).

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUE

A successful **chdir** returns zero. Otherwise, it returns -1 and sets *errno* as indicated below.

ERRORS

Chdir will fail and the current working directory will not change if one or more of the following are true:

[ENOTDIR]	A component of the pathname is not a directory.
[ENOENT]	The directory named does not exist.
[ENOENT]	The argument pathname is too long.
[EPERM]	The argument contains a byte with the high-order bit set.
[EACCES]	Search permission is denied for any component of the pathname.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

NAME

chmod – change mode of file

USAGE

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

fchmod(*fd*, *mode*)
char **path*;
int *fd*, *mode*;

DESCRIPTION

The **chmod** system call changes the mode of the file named by *path* to *mode*. **Fchmod** does the same thing to file descriptor *fd*. Modes are constructed from the logical OR of the following octal values.

04000 set user ID on execution
02000 set group ID on 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

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

Writing or changing the owner of a file turns off the set-user-ID and set-group-ID bits. This makes the system somewhat more secure by protecting set-user-ID (set-group-ID) files from remaining set-user-ID (set-group-ID) if they are modified.

NOTES

The DOMAIN System's single level store architecture requires that all filesystem objects be readable in order to be writable or executable. Since write-only or execute-only files would be unusable in DOMAIN/IX, modes that specify 02 (write-only) or 01 (execute-only) are ORed with 0400 to force read permission. This applies to the owner, group, and world portions of the mode word. For example, if mode 0631 were specified, the mode applied to the file would actually be 0675.

RETURN VALUE

A successful call to either **chmod** or **fchmod** returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Chmod will fail and the file mode will be unchanged if:

[EPERM]	The argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The pathname is too long.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

Fchmod will fail if:

[EBADF]	The descriptor is not valid.
[EINVAL]	*Fd* refers to a socket, not to a file.
[EROFS]	The file resides on a read-only file system.

RELATED INFORMATION

open(2), chown(2)

NAME

chown – change owner or group of a file

USAGE

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

fchown(*fd*, *owner*, *group***)**
int *fd*, *owner*, *group*;

DESCRIPTION

Chown (fchown) sets the *owner* and *group* of the object specified by *path* (or file descriptor *fd*). Only the super-user may execute this call.

On some systems, **chown** clears the set-user-ID and set-group-ID bits on the file to prevent accidental creation of set-user-ID and set-group-ID programs owned by the super-user.

Fchown is particularly useful when used in conjunction with the file-locking primitives (see **flock**(2)).

You may set either the owner or the group ID without changing the other. Set the ID you do not want to change to -1.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Chown will fail and the file will be unchanged if:

[EINVAL]	The argument path does not refer to a file.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The argument pathname is too long.
[EPERM]	The argument contains a byte with the high-order bit set.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EROFS]	The named file resides on a read-only file system.

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

[ELOOP] The call encountered too many symbolic links in translating the pathname.

Fchown will fail if:

[EBADF] *Fd* does not refer to a valid descriptor.

[EINVAL] *Fd* refers to a socket, not a file.

RELATED INFORMATION

chmod(2), **flock**(2)

NAME

close – delete a descriptor

USAGE

close(*d*)
int *d*;

DESCRIPTION

Close deletes descriptor *d* from the per-process object reference table. If this is the last reference to the underlying object, then the object will be deactivated. For example, on the last close of a file the current seek pointer associated with the file is lost; on the last close of a **socket**(2), the associated naming information and any queued data are discarded; on the last close of a file holding an advisory lock, the lock is released; see **flock**(2).

All of a process's descriptors close automatically upon an **exit**(2), but since there is a limit on the number of active descriptors per process, **close** is necessary for programs that use many descriptors.

When a process forks (see **fork**(2)), all descriptors held by the forked child process refer to the same objects as they did in the parent. If a new process is then run using **execve**(2), the process normally inherits these descriptors. Most of the descriptors can be rearranged with **dup2**(2) or deleted with **close** before the **execve** is attempted. However, if some of these descriptors are needed in case the **execve** fails, you must arrange to close them if the **execve** succeeds. Use **fcntl**(2) as shown here:

fcntl(*d*, **F_SETFD, 1**)

to arrange for descriptor *d* to be closed after a successful **execve**, and

fcntl(*d*, **F_SETFD, 0**)

to restore the default, i.e., that the descriptor does not close.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Close will fail if:

[EBADF] *d* is not an active descriptor.

RELATED INFORMATION

accept(2), flock(2), open(2), pipe(2), socket(2), socketpair(2), execve(2), fcntl(2)

NAME

connect – initiate a connection on a socket

USAGE

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

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The parameter *s* specifies a socket. If *s* is of the type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams will be sent; if it is of the type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way.

RETURN VALUE

A successful **connect** returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	This host cannot reach the network.

[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EWOULDBLOCK]	The socket is non-blocking, and the connection cannot be completed immediately.

RELATED INFORMATION

accept(2), select(2), socket(2)

NAME

creat – create a new file (obsolete)

USAGE

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

DESCRIPTION

This interface has been made obsolete by **open**(2).

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 created with *mode*, as modified by the process's mode mask (see **umask**(2)). Also see **chmod**(2) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged, but it is truncated to zero length. The file is also opened for writing, and its file descriptor is returned.

NOTES

The *mode* given is arbitrary; it need not allow writing. In the past, a *mode* that did not allow writing let programs construct a simple exclusive locking mechanism. This function has been replaced by the O_EXCL mode of **open**(2), and by the **flock**(2) facility.

The DOMAIN System's single level store architecture requires that all filesystem objects be readable by their owner. Since DOMAIN/IX does not allow write-only or execute-only files, modes 00100 (write only by owner) and 00200 (execute only by owner) are effectively ORed with 00400 to force read permission for the owner.

RETURN VALUE

A successful call returns a non-negative integer file descriptor that only permits writing. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Creat will fail and the file will not be created or truncated if one of the following occur:

[EPERM]	The argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	A needed directory does not have search permission.
[EACCES]	The file does not exist and the directory in which it would be created is not writable.
[EACCES]	The file exists, but it is unwritable.

[EISDIR]	The file is a directory.
[EMFILE]	There are already too many files open.
[EROFS]	The named file resides on a read-only file system.
[ENXIO]	The file is a character-special or block-special file, and the associated device does not exist.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	**Name** points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.
[EOPNOTSUPP]	The file was a socket (not currently implemented).

RELATED INFORMATION

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

NAME

default_acl – change default file protection environment

USAGE

#include <default_acl.h>

int default_acl(*switch*)
int *switch*;

DESCRIPTION

The DOMAIN/IX system call **default_acl** allows programs to change the default file protection environment between access mode and access control list (ACL). Values for the *switch* argument are defined in the include file *<default_acl.h>*. They are:

USE_DEFACL	Use the default ACL contained in the directory when creating a new file, pipe, or directory.
USE_MODE	Use the access mode supplied in the call, modified by the current **umask** value.
USE_DEFENV	Use the default for the environment in which the program is running. Unless the containing directory has a nil initial file acl (set using **sup**(8)), the default for programs running in an AEGIS environment is to use the intial file ACL. If the containing directory has a nil initial file acl, the default for programs running in an AEGIS environment is the same as for those running in a DOMAIN/IX environment. In all cases, the default for programs running in a DOMAIN/IX environment is to use the appropriate access mode.

RELATED INFORMATION

chmod(2) sup(8)

NAME

dup, dup2 – duplicate a descriptor

USAGE

newd = **dup**(*oldd*)
int *newd*, *oldd*;

dup2(*oldd*, *newd*)
int *oldd*, *newd*;

DESCRIPTION

Dup duplicates an existing object descriptor. The argument *oldd* is a small, non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by **getdtablesize**(2). The new descriptor *newd* returned by the call is the lowest-numbered descriptor that the process is not currently using.

The object that the descriptor refers to does not distinguish between references to *oldd* and *newd* in any way. Thus, if *newd* and *oldd* are duplicate references to an open file, **read**(2), **write**(2) and **lseek**(2) calls all move a single pointer into the file. If a separate pointer into the file is desired, you must create a different object reference to the file by issuing an additional **open**(2) call.

In the second form of the call, the value of the *newd* desired is specified. If this descriptor is already in use, the descriptor is deallocated first, as if a **close**(2) call had been done first.

RETURN VALUE

A successful call to either **dup** or **dup2** returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Dup and **dup2** fail if:

[EBADF]	*Oldd* or *newd* is not a valid active descriptor.
[EMFILE]	Too many descriptors are active.

RELATED INFORMATION

accept(2), **open**(2), **close**(2), **pipe**(2), **socket**(2), **socketpair**(2), **getdtablesize**(2)

NAME

execve – execute a file

USAGE

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

DESCRIPTION

Execve transforms the calling process into a new process. The new process is constructed from an ordinary file called the "new process file." This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. You can initialize additional pages with with zero data with the header.

An interpreter file begins with a line of the form

#! *interpreter*

where *interpreter* is the full pathname of the desired interpreter, for example

#! **/bin/sh**

When you execve an interpreter file, the system runs execve on the specified *interpreter*, giving it the name of the original file as an argument and shifting over the rest of the original arguments.

There is no return from a successful execve because the calling process's core image is overwritten by the new process.

The argument *argv* is an array of character pointers to null-terminated character strings that comprise an argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program (i.e., the last component of *name*).

The argument *envp* is also an array of character pointers to null-terminated strings. These strings pass information that is not in the form of direct arguments to the command.

Descriptors that were open in the calling process remain open in the new process, except those for which the close-on-exec flag is set; see close(2). Execve does not affect descriptors that remain open.

Ignored signals remain ignored across an execve, but signals that are caught are reset to their default values. The signal stack is reset to undefined; see sigvec(2) for more information.

Each process has "real" user and group IDs as well as "effective" user and group IDs. The real ID identifies the person using the system; the effective ID determines the user's access privileges. Execve 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 new process also inherits the following attributes from the calling process:

process ID	see **getpid** (2)
parent process ID	see **getppid** (2)
process group ID	see **getpgrp** (2)
access groups	see **getgroups** (2)
working directory	see **chdir** (2)
control terminal	see **tty** (4)
resource usages	see **getrusage**(2)
interval timers	see **getitimer** (2)
resource limits	see **getrlimit**(2)
file mode mask	see **umask** (2)
signal mask	see **sigvec** (2)

When the executed program begins, it is called as follows:

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

where *argc* is the number of elements in *argv* (the "arg count") and *argv* is the array of character pointers to the arguments themselves.

Envp is a pointer to an array of strings that constitutes the environment of the process. A pointer to this array is also stored in the global variable *environ*. Each string consists of a name, an "=", and a null-terminated value. The array of pointers ends with a null pointer. The shell passes an environment entry for each global shell variable defined when the program is called.

NOTES

If a program is "set-user-ID" to a non-super-user, but is executed when the real "user-ID" is "root," then the program has the powers of a super-user as well.

RETURN VALUE

A successful execve never returns. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[ENOENT]	One or more components of the new process file's pathname do not exist.
[ENOTDIR]	A component of the new process file is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file is not an ordinary file.
[EACCES]	The new process file mode denies execute permission.
[ENOEXEC]	The new process file has the appropriate access permission, but has an invalid magic number in its header.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing or reading by some process.
[ENOMEM]	The new process requires more virtual memory than is allowed by the imposed maximum (**getrlimit**(2)).
[E2BIG]	The number of bytes in the new process's argument list is larger than the system-imposed limit of {ARG_MAX} bytes.
[EFAULT]	The new process file is not as long as the size value indicated in its header.
[EFAULT]	*Path*, *argv*, or *envp* point to an illegal address.

RELATED INFORMATION

exit(2), **fork**(2), **execl**(3)

NAME

_exit – terminate a process

USAGE

_exit(*status*)
int *status*

DESCRIPTION

The **_exit** system call terminates a process with the following consequences:

- All of the descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a **wait** or is interested in the SIGCHLD signal, it is notified of the calling process's termination and the low-order eight bits of *status* are made available to it; as detailed in the entry for **wait**(2).
- The parent process ID of all of the calling process's existing child processes are also set to 1. This means that the initialization process (see **intro**(2)) inherits each of these processes as well.

Most C programs call the library routine **exit**(3), which performs clean-up actions in the standard I/O library before calling **_exit**.

RETURN VALUE

This call never returns.

RELATED INFORMATION

fork(2), **wait**(2), **exit**(3)

NAME

fcntl – file control

USAGE

#include <fcntl.h>

res = **fcntl(***fd*, *cmd*, *arg***)**
int *res*;
int *fd*, *cmd*, *arg*;

DESCRIPTION

Fcntl provides various types of control over file descriptors. Several varieties of *cmd* are provided, which operate on *fd* as follows.

F_DUPFD — Return a new descriptor that:

- is the lowest-numbered available descriptor greater than or equal to *arg*,
- references the same object as the original *fd*,
- shares the same file pointer if the object was a file,
- has the same access mode (read, write or read/write) as the original *fd*,
- has the same file-status flags (i.e., both file descriptors share the same file status flags),
- sets the close-on-exec flag associated with the new file descriptor to remain open across execve(2) system calls.

F_GETFD — Get the close-on-exec flag associated with the file descriptor *fd*. If the low-order bit is zero, the file will remain open across exec; otherwise, the file will close upon execution of exec.

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

F_GETFL — Get descriptor status flags, as described below.

F_SETFL — Set descriptor status flags.

F_GETOWN — Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values.

F_SETOWN — Set the process or process group to receive SIGIO and SIGURG signals; you can specify process groups by supplying a negative *arg*;

otherwise *arg* is interpreted as a process ID.

The flags for the F_GETFL and F_SETFL flags are as follows:

FNDELAY — Non-blocking I/O; if no data is available to a **read**(2) call, or if a **write**(2) operation would block, the call returns -1 and sets *errno* to the value EWOULDBLOCK.

FAPPEND — Force each write to append at the end of file (corresponds to the O_APPEND flag of **open**(2).)

RETURN VALUE

The value returned upon successful completion depends on *cmd* as follows:

F_DUPFD — returns a new file descriptor.

F_GETFD — returns the value of the close-on-exec flag (only the low-order bit is defined).

F_GETFL — returns the values of the applicable flags.

F_GETOWN — returns the value of file descriptor owner.

All others — return some value other than -1

Otherwise, **fcntl** returns -1 and sets *errno* as indicated below.

ERRORS

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

[EBADF] — *Fd* is not a valid open file descriptor.

[EMFILE] — *Cmd* is F_DUPFD and the maximum allowed number of file descriptors are currently open.

[EINVAL] — *Cmd* is F_DUPFD and *arg* is negative or greater than the maximum allowable number (see **getdtablesize**(2)).

RELATED INFORMATION

close(2), **execve**(2), **getdtablesize**(2), **open**(2), **sigvec**(2)

NAME

flock – place or remove an advisory lock on an open file

USAGE

#include <sys/file.h>

```
#define    LOCK_SH   1    /* shared lock */
#define    LOCK_EX   2    /* exclusive lock */
#define    LOCK_NB   4    /* don't block when locking */
#define    LOCK_UN   8    /* unlock */
```

flock(*fd*, *operation*)
int *fd*, *operation*;

DESCRIPTION

Flock applies or removes an advisory lock on the file identified by the descriptor *fd*. A lock is applied by specifying an *operation* parameter which is the (inclusive) OR of LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock, *operation* should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency. (Processes may still access files without using advisory locks, and this may result in inconsistencies).

The locking mechanism allows two types of locks: "shared" locks and "exclusive" locks. Multiple shared locks may be applied to a file at any time. At no time are multiple exclusive locks, or a combination of shared and exclusive locks, allowed on a file.

A shared lock may be upgraded to an exclusive lock (or an exclusive lock turned into a shared lock) by specifying the appropriate lock type; this releases the previous lock and applies the new one.

Requesting a lock on an object that is already locked normally causes the caller to blocked until the lock can be acquired. If LOCK_NB is included in *operation*, such calls will fail and return the error EWOULDBLOCK instead.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through **dup**(2) or **fork**(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

Processes that are blocked waiting for a lock may be awakened by signals.

All processes that use advisory locks on a given file must be running on the same node.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The **flock** call fails if:

[EWOULDBLOCK]	The file is locked and the LOCK_NB option was specified.
[EBADF]	The argument *fd* is an invalid descriptor.
[EINVAL]	The argument *fd* refers to an object other than a file.

RELATED INFORMATION

open(2), **close**(2), **dup**(2), **execve**(2), **fork**(2)

NAME

fork – create a new process

USAGE

pid = **fork**()
int *pid*;

DESCRIPTION

Fork creates a new process that is a descendant of the process that calls **fork**. With the following exceptions, the new (child) process is an exact copy of the calling (parent) process.

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent. A **lseek**(2) on a descriptor in the child process, for example, can affect a subsequent **read**(2) or **write**(2) by the parent. Shells copy descriptors in this way to establish standard input and output for newly created processes, as well as to set up pipes.
- The child process's resource utilizations are set to zero; see **getrlimit**(2).

NOTES

On DOMAIN systems, **fork** may produce unexpected or undesired results when called from an **mbx** server process, or form a process using **gpr** or **gpio**.

RETURN VALUE

Upon successful completion, **fork** returns zero to the child process and returns the child's process ID to the parent process. Otherwise, -1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

ERRORS

Fork will fail and no child process will be created if either of the following is true:

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

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

RELATED INFORMATION

execve(2), wait(2)

NAME

fsync – synchronize a file's in-core state with that on disk

USAGE

fsync(*fd*)

int *fd*;

DESCRIPTION

Fsync causes all modified data and attributes of the object referenced by *fd* to be moved to a permanent (typically disk) storage device. This normally force-writes all modified copies of buffers for the associated file.

Fsync should be used by programs that require a file to be in a known state; for example in building a simple transaction facility.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The **fsync** fails if:

[EBADF] *Fd* is not a valid descriptor.

[EINVAL] *Fd* refers to a socket, not to a file.

NAME

getdtablesize – get descriptor table size

USAGE

nds = **getdtablesize()**
int *nds*;

DESCRIPTION

Each process has a fixed size descriptor table that is guaranteed to have at least 20 slots. The entries in the descriptor table are all small integers. The lowest-numbered descriptor is zero.

RETURN VALUE

The call **getdtablesize** returns a non-negative integer (the size of the descriptor table).

RELATED INFORMATION

close(2), **dup**(2), **open**(2)

NAME

getgid, getegid – get group identity

USAGE

gid = **getgid()**
int *gid*;

egid = **getegid()**
int *egid*;

DESCRIPTION

Getgid reports the real group ID of the current process; **getegid** reports the effective group ID.

The real group ID is set at log-in time. The effective group ID determines additional access permission during execution of a "set-group-ID" process. It is for such processes that **getgid** is most useful.

RETURN VALUE

Getgid reports the process's real group ID. **Getegid** reports the process's effective group ID.

RELATED INFORMATION

getuid(2), **setregid**(2), **setgid**(3)

NAME

getgroups – get group access list

USAGE

#include <sys/param.h>

ngroups = **getgroups(** *gidsetlen*, *gidset*)
int *ngroups*, *gidsetlen*, **gidset*;

DESCRIPTION

Getgroups obtains the current group access list of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in **gidset**. **Getgroups** returns the actual number of groups returned in *gidset*. No more than NGROUPS, as defined in *<sys/param.h>*, will ever be returned.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The possible errors for **getgroup** are:

[EINVAL] The argument *gidsetlen* is smaller than the number of groups in the group set.

[EFAULT] The arguments *ngroups* or *gidset* specify invalid addresses.

RELATED INFORMATION

setgroups(2), initgroups(3X)

NAME

gethostid, sethostid – get/set unique identifier of current host

USAGE

hostid = **gethostid()**
int *hostid*;

sethostid(*hostid***)**
int *hostid*;

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor. This identifier is intended to be unique among all UNIX systems in existence; it is normally a DARPA Internet address for the local machine. Use of this call is limited to the super-user, and typically occurs only at boot time.

Gethostid returns the 32-bit identifier for the current processor.

RETURN VALUE

Upon successful execution, **gethostid** returns the 32-bit identifier for the current processor.

RELATED INFORMATION

hostid(1), gethostname(2)

NAME

gethostname, **sethostname** – get/set name of current host

USAGE

gethostname(*name*, namelen)
char **name*;
int *namelen*;

sethostname(*name*, namelen)
char **name*;
int *namelen*;

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by **sethostname**. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated, unless insufficient space is provided in *namelen*.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. Use of **sethostname** is restricted to the super-user. It is typically used only when the system is booted.

NOTES

On some systems, host names are limited to 255 characters. DOMAIN/IX has no such limitation.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

These calls may return one or more of the following errors:

[EFAULT] The *name* or *namelen* parameter gave an invalid address.

[EPERM] The caller was not the super-user.

RELATED INFORMATION

gethostid(2)

NAME

getitimer, setitimer – get/set value of interval timer

USAGE

```
#include <sys/time.h>

#define ITIMER_REAL      0   /* real time intervals */
#define ITIMER_VIRTUAL   1   /* virtual time intervals */
#define ITIMER_PROF      2   /* user and system virtual time */

getitimer(which, value)
int which;
struct itimerval *value;

setitimer(which, value, ovalue)
int which;
struct itimerval *value, *ovalue;
```

DESCRIPTION

The system provides each process with three interval timers, defined in <sys/time.h>. The **getitimer** call returns the current value for the timer specified in the argument *which*, while the **setitimer** call sets the value of a timer. (**Getitimer** may also return the previous value of the timer.)

A timer value comes from the **itimerval** structure:

```
struct itimerval {
        struct  timeval it_interval;   /* timer interval */
        struct  timeval it_value;      /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to zero disables a timer. Setting *it_interval* to zero causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock (4 µseconds on DOMAIN systems) are rounded up to this resolution.

The ITIMER_REAL timer decrements in real time and delivers a SIGALRM signal when it expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing, and delivers a SIGVTALRM signal when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

NOTES

Three macros for manipulating time values are defined in <*sys/time.h*>. **Timerclear** sets a time value to zero, **timerisset** tests if a time value is non-zero, and **timercmp** compares two time values (>= and <= do not work with this macro).

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The possible errors are:

[EFAULT] The *value* structure specified a bad address.

[EINVAL] A *value* structure specified that a time was too large to be handled.

RELATED INFORMATION

sigvec(2), gettimeofday(2) select(2)

NAME

getpagesize – get system page size

USAGE

pagesize = getpagesize()
int *pagesize*;

DESCRIPTION

Getpagesize returns the number of bytes in a page, which is the granularity of many of the memory management calls.

The page size is a system page size, which may not be the same as the underlying hardware page size.

RETURN VALUE

This call returns the number of bytes in a page.

RELATED INFORMATION

sbrk(2), pagesize(1)

NAME

getpeername – get name of connected peer

USAGE

getpeername(*s*, *name*, *namelen***)**
int *s*;
struct sockaddr **name*;
int **namelen*;

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space *name* points to. On return, it contains the actual size of the name returned (in bytes).

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient system resources were available.
[EFAULT]	The *name* parameter points to memory that is not in a valid part of the process address space.

RELATED INFORMATION

bind(2), socket(2), getsockname(2)

NAME

getpgrp – get process group

USAGE

pgrp = **getpgrp(***pid***)**
int *pgrp*, *pid*;

DESCRIPTION

Getpgrp returns the process group of the specified process. If *pid* is zero, then the call applies to the current process.

Process groups are used to distribute signals, and by terminals to arbitrate requests for their input. Processes that have the same process group as the terminal are foreground and may read, while others will block and send a signal if they attempt to read.

Programs like **csh**(1) use this call to create process groups used in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in **tty**(4) are used to get or set the process group of the control terminal.

RELATED INFORMATION

setpgrp(2), getuid(2), tty(4)

NAME

getpid, getppid – get process identification

USAGE

pid = **getpid()**
long *pid*;

ppid = **getppid()**
long *ppid*;

DESCRIPTION

Getpid returns *pid*, the process ID of the current process. It is used most often with the host identifier **gethostid(2)** to generate uniquely-named temporary files.

Getppid returns *ppid*, the process ID of the parent of the current process.

RETURN VALUE

A successful **getpid** returns the process ID of the current process.

RELATED INFORMATION

gethostid(2)

NAME

getpriority, **setpriority** – get/set program scheduling priority

USAGE

```
#include <sys/resource.h>
#define PRIO_PROCESS      0    /* process */
#define PRIO_PGRP         1    /* process group */
#define PRIO_USER         2    /* user id */

prio = getpriority( which, who)
int prio, which, who;

setpriority( which, who, prio)
int which, who, prio;
```

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* can be obtained with the **getpriority** call and set with the **setpriority** call. The *which* parameter can be one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER. The *who* parameter is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). **Prio** is a value in the range -20 to 20. The default priority is zero; lower priorities cause more favorable scheduling.

The **getpriority** call returns the highest priority (lowest numerical value) held by any of the specified processes. The **setpriority** call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

RETURN VALUE

Since **getpriority** can legitimately return the value -1, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a returned -1 is an indication of error or a legitimate priority value.

A successful **setpriority** call returns zero. A failed **setpriority** call returns -1 and sets *errno* as indicated below.

ERRORS

Getpriority and **setpriority** may return one of the following errors:

[ESRCH] No process was located using the *which* and *who* values specified.

[EINVAL] *Which* was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, **setpriority** may fail with one of the following errors returned:

[EACCES] A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

[EACCES] A non super-user attempted to change a process priority to a negative value.

RELATED INFORMATION

nice(1), fork(2), renice(8)

NAME

getrlimit – control maximum system resource consumption

USAGE

#include <sys/time.h>
#include <sys/resource.h>

getrlimit(*resource*, *rlp***)**
int *resource*;
struct rlimit **rlp*;

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the **getrlimit** call.

The *resource* parameter is one of the following:

RLIMIT_CPU	Maximum amount of CPU time (in milliseconds) to be used by each process. Currently, this is always RLIMIT_INFINITY.
RLIMIT_FSIZE	Largest size, in bytes, of any single file that may be created.
RLIMIT_DATA	Maximum size, in bytes, of the data segment for a process; this defines how far a program may extend its break with the **sbrk**(2) system call.
RLIMIT_STACK	Maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended.
RLIMIT_CORE	Largest size, in bytes, of a *core* file that may be created. Currently, this is always 0.
RLIMIT_RSS	Maximum size, in bytes, to which a process's resident set size may grow. Currently, this is always RLIMIT_INFINITY. A limit is imposed on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes which are exceeding their declared resident set size.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the CPU time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The *rlimit* structure is used to specify the hard and soft limits on a resource,

struct rlimit {

```
        int     rlim_cur;       /* current (soft) limit */
        int     rlim_max;       /* hard limit */
};
```

An "infinite" value for a limit is defined as RLIMIT_INFINITY (0x7fffffff).

The system refuses to extend data or stack space when the limits would be exceeded in the normal way: a **break**(2) call fails if the data space limit is reached, or the process is killed when the stack limit is reached (since the stack cannot be extended, there is no way to send a signal).

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

[EFAULT] The address specified for *rlp* is invalid.

RELATED INFORMATION

csh(1), quota(2)

NAME

getrusage – get information about resource utilization

USAGE

```
#include <sys/time.h>
#include <sys/resource.h>

#define  RUSAGE_SELF         0  /* calling process */
#define  RUSAGE_CHILDREN -1  /* terminated child processes */

getrusage( who, rusage)
int who;
struct rusage *rusage;
```

DESCRIPTION

The **getrusage** call returns information describing the resources used by the current process or all of its terminated child processes. The *who* parameter is one of RUSAGE_SELF and RUSAGE_CHILDREN. If *rusage* is non-zero, the buffer it points to will be occupied by the following structure:

```
struct  rusage  {
        struct timeval ru_utime;       /* user time used */
        struct timeval ru_stime;       /* system time used */
        int  ru_maxrss;
        int  ru_ixrss;                 /* integral shared memory size */
        int  ru_idrss;                 /* integral unshared data size */
        int  ru_isrss;                 /* integral unshared stack size */
        int  ru_minflt;                /* page reclaims */
        int  ru_majflt;                /* page faults */
        int  ru_nswap;                 /* swaps */
        int  ru_inblock                /* block input operations */
        int  ru_oublock;               /* block output operations */
        int  ru_msgsnd;                /* messages sent */
        int  ru_msgrcv;                /* messages received */
        int  ru_nsignals;              /* signals received */
        int  ru_nvcsw;                 /* voluntary context switches */
        int  ru_nivcsw;                /* involuntary context switches */
};
```

Currently, only the following fields are meaningful to DOMAIN/IX operations:

ru_utime Total amount of time spent executing in user mode.

ru_majflt Number of page faults serviced that required I/O activity.

ru_nsignals	Number of signals delivered.

The remaining fields are returned as zero. Moreover, the only information returned about child processes is user time (*ru_time*); all other fields are returned as zero.

CAUTIONS

There is no way to obtain information about a child process that has not yet terminated.

RELATED INFORMATION

gettimeofday(2)
wait(2)

NAME

getsockname – get socket name

USAGE

getsockname(*s*, *name*, *namelen***)**
int *s*, *namelen*;
struct sockaddr **name*;

DESCRIPTION

Getsockname returns the current *name* for the specified socket *s*. The *namelen* parameter should be initialized to indicate the amount of space that *name* points to. On return, it contains the size, in bytes, of the name.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient system resources were available.
[EFAULT]	The *name* parameter points to memory that isn't in a valid part of the process's address space.

RELATED INFORMATION

bind(2), socket(2)

NAME

getsockopt, setsockopt – get/set options on sockets

USAGE

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

getsockopt(*s*, *level*, *optname*, *optval*, *optlen*)
int *s*, *level*, *optname*;
char **optval*;
int **optlen*;

setsockopt(*s*, *level*, *optname*, *optval*, *optlen*)
int *s*, *level*, *optname*;
char **optval*;
int **optlen*;

DESCRIPTION

Getsockopt and **setsockopt** manipulate options associated with socket *s*. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see **getprotoent**(3N).

The parameters *optval* and *optlen* are used to access option values for **setsockopt**. For **getsockopt**, *optval* and *optlen* identify a buffer in which the value for the requested option(s) is to be returned. For **getsockopt**, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be designated as zero.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sys/socket.h>* contains definitions for socket level options; see **socket**(2).

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

These calls succeed unless:

[EBADF]	The argument s is not a valid descriptor.
[ENOTSOCK]	The argument s is a file, not a socket.
[ENOPROTOOPT]	The option is unknown.
[EFAULT]	Options are not in a valid part of process address space.

RELATED INFORMATION

socket(2), **getprotoent**(3N)

NAME

gettimeofday, settimeofday – get/set date and time

USAGE

#include <sys/time.h>

gettimeofday(*tp*, *tzp*)
struct timeval **tp*;
struct timezone **tzp*;

settimeofday(*tp*, *tzp*)
struct timeval **tp*;
struct timezone **tzp*;

DESCRIPTION

Gettimeofday returns the system's idea of the current Greenwich time and the current time zone. Time returned is expressed in seconds and microseconds since midnight, January 1, 1970.

The structures pointed to by *tp* and *tzp* are defined in *<sys/time.h>* as:

```
struct timeval {
        u_long tv_sec;          /* seconds since Jan. 1, 1970 */
        long   tv_usec;         /* and microseconds */
};

struct timezone {
        int    tz_minuteswest;        /* of Greenwich */
        int    tz_dsttime;            /* type of dst correction to apply */
};
```

The **timezone** structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Settimeofday is illegal on DOMAIN/IX systems. Any attempt to set the time returns an error.

NOTES

Time is not correct to the microsecond values.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] An argument address referred to invalid memory.

[EPERM] On DOMAIN/IX Systems, an attempt was made to use **settimeofday**. On other systems, an unprivileged process attempted use **settimeofday**.

RELATED INFORMATION

date(1), ctime(3)

NAME

getuid, geteuid – get user identity

USAGE

uid = **getuid()**
int *uid*;

euid = **geteuid()**
int *euid*;

DESCRIPTION

Getuid returns the real user ID of the current process, **geteuid** the effective user ID.

The real user ID (*uid*) identifies the account that is logged in. The effective user ID (*euid*) gives the process additional permissions during execution of "set-user-ID" mode processes, which use **getuid** to determine the real user-ID of the process which invoked them.

RETURN VALUE

If successful, these calls return the real user ID and effective user ID, respectively, of the current process.

RELATED INFORMATION

getgid(2), setreuid(2)

NAME

ioctl – control device

USAGE

#include <sys/ioctl.h>

ioctl(*d*, *request*, *argp*)
int *d*, *request*;
char **argp*;

DESCRIPTION

Ioctl calls perform a variety of functions on open descriptors. They are typically used to control the characteristics of character-special files (e.g., terminals).

An **ioctl** *request* specifies whether the argument is an "in" parameter or an "out" parameter, as well as the size of the argument *argp* in bytes. Macros and definitions used in specifying an **ioctl** request are in the file <**sys/ioctl.h**>.

NOTES

When **ioctl** is used in programs that deal with DOMAIN System Display Manager pads, setting the mode to RAW has the immediate effect of putting the pad into raw mode. Other **ioctl** modes have no effect, but are stored and will be inherited by the **vt100** program if it is subsequently invoked in that pad.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[EBADF] *D* is not a valid descriptor.

[ENOTTY] *D* is not associated with a character-special device.

[ENOTTY] The specified request does not apply to the kind of object that the descriptor *d* references.

[EINVAL] *Request* or *argp* is not valid.

RELATED INFORMATION

execve(2), fcntl(2)

NAME

kill – send signal to a process

USAGE

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

DESCRIPTION

Kill sends signal *sig* to the process identified by the process number *pid*. *Sig* may be one of the signals specified in **sigvec(2)**, or it may be zero, in which case error checking (e.g., to see if the process specified by *pid* exists) is performed but no signal is actually sent.

Both the sending and receiving processes must have the same effective user ID. The only exception is the signal SIGCONT, which **kill** can always send to any child or grandchild of the current process. In all other cases, the use of **kill** is restricted to the super-user.

If the process number is zero, *sig* is sent to all other processes in the sender's process group; this is a variant of **killpg(2)**.

If the process number is -1 and the user is the super-user the signal is sent to all processes running on the machine, with the exception of system processes and the process sending the signal.

Processes may send signals to themselves.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Kill will fail and no signal will be sent in the following instances:

[EINVAL] *Sig* is not a valid signal number.

[ESRCH] No process can be found with the specified *pid.*

[EPERM] The sending process is not the super-user and its effective user ID does not match the effective user-ID of the receiving process.

RELATED INFORMATION

getpid(2), getpgrp(2), killpg(2), sigvec(2)

NAME

killpg – send signal to a process group

USAGE

killpg(*pgrp, sig***)**
int *pgrp, sig***;**

DESCRIPTION

Killpg sends the signal *sig* to the process group *pgrp*. *Sig* must be one of the signals defined in **sigvec(2)**.

The sending process and all processes in the process group must have the same effective user ID. The only exception is the signal SIGCONT, which **killpg** may always send to any child or grandchild of the current process. In all other cases, use of **killpg** is restricted to the super-user.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Killpg will fail and no signal will be sent in any of the following cases:

[EINVAL] *Sig* is not a valid signal number.

[ESRCH] No process was found with the specified *pid*.

[EPERM] The sending process is not the super-user and one or more of the target processes has a different effective user ID than the sending process.

RELATED INFORMATION

kill(2), getpgrp(2), sigvec(2)

NAME

link – make a hard link to a file

USAGE

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

DESCRIPTION

Link creates a hard link to *name1*; the new link takes the name *name2*. *Name1* must exist before the call to **link** is made.

Both *name1* and *name2* must be in the same file system. On DOMAIN Systems, *name1* cannot be a directory. Both the old and the new link have the same rights to the underlying object.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[EPERM]	Either pathname contains a byte with the high-order bit set.
[ENOENT]	Either pathname is too long.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *name1* does not exist.
[EEXIST]	The link named by *name2* already exists.
[EPERM]	The file named by *name1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *name2* and the file named by *name1* are on different file systems.
[EACCES]	The requested link requires writing in a directory mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	One of the pathnames specified lies outside the process's allocated

address space.

[ELOOP] The call encountered too many symbolic links in translating the pathname.

RELATED INFORMATION

symlink(2), unlink(2)

NAME

listen – listen for connections on a socket

USAGE

listen(*s*, *backlog*)
int *s*, *backlog*;

DESCRIPTION

To accept connections, a socket is created with **socket**(2), a backlog for incoming connections is specified with **listen**(2), and the connections are accepted with **accept**(2).

The *backlog* parameter defines the maximum length of the queue of pending connections. If a connection request arrives and the queue is full, the client will receive the error ECONNREFUSED.

NOTES

The maximum value for *backlog* is five.

The **listen** call applies only to sockets of the type SOCK_STREAM or SOCK_PKTSTREAM.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket type is unsupported by **listen** (it is not one of type SOCK_STREAM or SOCK_PKTSTREAM).

RELATED INFORMATION

accept(2), connect(2), socket(2)

NAME

lseek – move read/write pointer

USAGE

```
#define L_SET    0  /* set the seek pointer */
#define L_INCR   1  /* increment the seek pointer */
#define L_XTND   2  /* extend the file size */
```

pos = **lseek(** *d*, *offset*, *whence***)**
int *pos*;
int *d*, *offset*, *whence*;

DESCRIPTION

The descriptor *d* refers to a file or device open for reading and/or writing. Lseek sets the file pointer of *d* as follows:

- If *whence* is L_SET, the pointer is set to *offset* bytes.
- If *whence* is L_INCR, the pointer is set to its current location plus *offset*.
- If *whence* is L_XTND, the pointer is set to the size of the file plus *offset*.

Upon successful completion, lseek returns the resulting pointer location, measured in bytes from the beginning of the file.

The *whence* values are defined in <*sys/file.h*>.

NOTES

If lseek goes far beyond the end of a file, and then writes, it creates a gap that occupies no physical space and reads as zeros.

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

RETURN VALUE

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

ERRORS

Lseek will fail and the file pointer will remain unchanged if:

[EBADF] *D* is not an open file descriptor.

[ESPIPE] *D* is associated with a pipe or a socket.

[EINVAL] *Whence* is not a proper value.

[EINVAL] The resulting file pointer would be negative.

RELATED INFORMATION

dup(2), open(2)

NAME

mkdir – make a directory file

USAGE

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

DESCRIPTION

Mkdir creates a new directory file with the name *path*. *Mode* sets the new directory's mode. (The protection part of the mode is modified by the process's mode mask; see **umask**(2)).

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to that of the parent directory in which it is created.

The low-order 9 bits of *mode* are modified by the process's file mode creation mask; all bits set in the process's file mode creation mask are cleared. See **umask**(2).

NOTES

The DOMAIN System's single level store architecture requires that all filesystem objects be readable in order to be writable or executable. Since write-only or execute-only files would be unusable in DOMAIN/IX, modes that specify 02 (write-only) or 01 (execute-only) are ORed with 0400 to force read permission. This applies to the owner, group, and world portions of the *mode* word. For example, if mode 0631 were specified, the mode applied to the file would actually be 0675.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Mkdir will fail and no directory will be created if:

[EPERM] The *path* argument contains a byte with the high-order bit set.

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

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

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

[EEXIST] The named file already exists.

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

[ELOOP] The call encountered too many symbolic links in translating the pathname.

[EIO] An I/O error occurred while the call was writing to the file system.

RELATED INFORMATION

chmod(2), stat(2), umask(2)

NAME

mknod – make a special file

USAGE

mknod(*path*, *mode*, *dev*)
char **path*;
int *mode*, *dev*;

DESCRIPTION

Mknod creates a new file whose name is *path*. *Mode* sets the mode of the new file, including the special file bits. (The protection part of the mode is modified by the process's mode mask; see **umask(2)**).

If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Use of **mknod** is limited to the super-user.

Mode is interpreted as follows:

0170000	file type; one of the following:	
	0010000	fifo special
	0040000	directory
	0100000	ordinary file
	0000000	ordinary file
0004000	set user ID on execution	
0002000	set group ID on execution	
0000777	access permissions; constructed from the following	
	0000400	read by owner
	0000200	write by owner
	0000100	execute (search on directory) by owner
	0000070	read, write, execute (search) by group
	0000007	read, write, execute (search) by others

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

Values of **mode** other than those above are undefined, and should not be used. The low-order 9 bits of **mode** are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See **umask(2)**.

NOTES

The DOMAIN System's single level store architecture requires that all filesystem objects be readable in order to be writable or executable. Since write-only or execute-only files would be unusable in DOMAIN/IX, modes that specify 02 (write-only) or 01 (execute-only) are ORed with 0400 to force read permission. This applies to the owner, group, and world portions of the *mode* word. For example, if mode 0631 were specified, the mode applied to the file would actually be 0675.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Mknod will fail if:

[EPERM]	The process's effective user ID is not super-user.
[EPERM]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The named file resides on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

RELATED INFORMATION

chmod(2), stat(2), umask(2)

NAME

mount, **umount** – mount or remove file system

USAGE

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

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

DESCRIPTION

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

The *name* must not already exist; it is created by the **mount** call and exists only for the duration of the file system mount.

The *rwflag* argument controls write access to the *special* file system. If *rwflag* is 0, writing is allowed. If it is non-zero, writing is prohibited. Physically write-protected file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

The **umount** call announces to the system that the *special* file no longer contains a removable file system. The associated file is removed.

RETURN VALUE

The **mount** call returns 0 or –1.

0 Specified operation was successful.

–1 The *special* file is inaccessible, already mounted, or not an appropriate file; *name* does not exist or is in use; or there are already too many file systems mounted.

The **umount** call returns 0 or –1.

0 Specified operation was successful.

–1 The *special* file is inaccessible or does not have a mounted file system, or there are active files in the mounted file system.

ERRORS

Under the following conditions, **mount** fails:

[NODEV] *Special* does not exist.

[ENOTBLK] *Special* is not a block device.

[ENXIO] The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).

[EPERM] The pathname contains a character with the high-order bit set.

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

[EROFS] *Name* resides on a read-only file system.

[EBUSY] *Name* already exists.

[EBUSY] No space remains in the mount table.

[EBUSY] The super-block for the file system had a bad magic number or an out-of-range block size.

[EBUSY] Not enough memory was available to read the cylinder group information for the file system.

[EBUSY] An I/O error occurred while reading the super block or cylinder group information.

Under the following conditions, **umount** fails:

[NODEV] *Special* does not exist.

[ENOTBLK] *Special* is not a block device.

[ENXIO] The major device number of *special* is out of range (no device driver exists for the associated hardware).

[EINVAL] The requested device is not in the mount table.

[EBUSY] A process is holding a reference to a file located on the file system.

Note that the error codes are not always informative. Many types of errors (e.g., no space in the mount table, not enough memory, etc.) return the same value (e.g., EBUSY) to the caller.

RELATED INFORMATION

mkdisk(8), mount(8), umount(8)

NAME

open – open a file for reading or writing, or create a new file

USAGE

#include <sys/file.h>

open(*path*, *flags*, *mode***)**
char **path*;
int *flags*, *mode*;

DESCRIPTION

Open opens the file named by *path* for reading and/or writing, as specified by the *flags* argument and returns a descriptor for that file. The *flags* argument may indicate that the file is to be created if it does not already exist (the O_CREAT flag). In this case, the file is created with mode *mode*, as described in **chmod**(2) and as modified by the process's umask value (see **umask**(2)).

Path is the address of a null-terminated string of ASCII characters representing a pathname. The flags are formed from the logical OR of the following values:

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NDELAY	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to zero
O_EXCL	error if create and file exists

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT and the file already exists, the open returns an error. This can be used to implement a simple exclusive access locking mechanism. If the O_NDELAY flag is specified and the open call would result in the process being blocked for some reason (e.g., waiting for carrier on a dial-up line), the **open** returns immediately. The first time the process attempts to perform I/O on the open file, it will block.

NOTES

The DOMAIN System's single level store architecture requires that all filesystem objects be readable in order to be writable or executable. Since write-only or execute-only files would be unusable in DOMAIN/IX, modes that specify 02 (write-only) or 01 (execute-only) are ORed with 0400 to force read permission. This applies to the owner, group, and world portions of the *mode* word. For example, if mode

0631 were specified, the mode applied to the file would actually be 0675.

No process may have more than {OPEN_MAX} file descriptors open simultaneously.

RETURN VALUE

Upon successful completion, a non-negative integer file descriptor is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The new descriptor is set to remain open across execve system calls; see **close**(2). A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[EPERM] The pathname contains a character with the high-order bit set.

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

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

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

[EACCES] The required permissions (for reading and/or writing) are denied for the named flag.

[EISDIR] The named file is a directory, and the arguments specify that it is to be opened for writing.

[EROFS] The named file resides on a read-only file system, and the file is to be modified.

[EMFILE] {OPEN_MAX} (usually 20) file descriptors are currently open.

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

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed, and the **open** call requests write access.

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

[ELOOP] The call encountered too many symbolic links in translating the pathname.

[EEXIST] O_EXCL was specified and the file exists.

[ENXIO] The O_NDELAY flag is given, and the file is a communications device on which no carrier is present.

RELATED INFORMATION

chmod(2), close(2), dup(2), lseek(2), read(2), write(2), umask(2)

NAME

pipe – create an interprocess communication channel

USAGE

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**(2) and **write**(2) operations. When the pipe is written using the descriptor *fildes*[**1**], up to 5120 bytes of data are buffered before the writing process is suspended. A **read**(2) using the descriptor *fildes*[**0**] will pick up the data.

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

The shell has a syntax that allows users to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (one with no buffered data and no writers) return an end-of-file.

Attempts to write to a pipe that has no readers will generate a SIGPIPE signal.

NOTES

Deadlock will occur if more than 5120 bytes are necessary in any pipe among a loop of processes.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The **pipe** call will fail if:

[EMFILE] Too many descriptors are active.

[EFAULT] The *fildes* buffer is in an invalid area of the process's address space.

RELATED INFORMATION

sh(1), **read**(2), **write**(2), **fork**(2), **socketpair**(2)

NAME

ptrace – process trace

USAGE

#include <signal.h>

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

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process 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 **sigvec**(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:

Note: Where two numbers are associated with a *request* (an artifact of implementations with separate instruction and data space), either number may be used.

Request zero can only be used in the child. Non-zero *request*s can only be used by the parent. For each non-zero *request*, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

0 Child trace flag. This is the only request that can be issued by the child. It stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by any *func*; argument associated with a **signal**(2) call in the child. The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

1, 2 return the word at location *addr* in the address space of the child. On DOMAIN Systems, either request **1** or request **2** may be used with identical results. If *addr* is not the start address of a word, a value of –1 is returned to the parent process and the parent's *errno* is set to *EIO*.

3 return the word at offset *addr* into the child's USER area in the system's address space (see <*sys/user.h*>) to the parent process. (Only 16 bits can be read.) If *addr* is not the start address of a word or is outside the USER area, a value of -1 is returned to the parent process and the parent's *errno* is set to *EIO*.

4, 5 write the value given by the *data* argument into the address space of the child at location *addr*. Upon successful completion, the value written into the address space of the child is returned to the parent. If *addr* is a location in a pure procedure space and another process is executing in that space, or if *addr* is not the start address of a word, these requests will fail, a value of –1 will be returned to the parent process, and the parent's *errno* will be set to *EIO*.

6 write one of the following entries, where *data* is a 16-bit value to be written and *addr* is the location of the entry in the child's USER area:

M68xxx processor registers (A0-A7, D0-D7).

The condition codes (bits 0-7) of the Processor Status Word

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

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

9 This request sets the trace bit in the Processor Status Word of the child (bit 15 on M68xxx processors) and then executes the same steps as listed above for request 7. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child. The trace bit is turned off after interrupt.

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

NOTES

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

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

[EINVAL]	The request code is invalid.
[EINVAL]	The specified process does not exist.
[EINVAL]	The given signal number is invalid.
[EFAULT]	The specified address is out of bounds.
[EPERM]	The specified process cannot be traced.

RELATED INFORMATION

wait(2), sigvec(2)

NAME

read, readv – read input

USAGE

cc = **read(** *d*, *buf*, *nbytes***)**
int *cc*, *d*;
char **buf*;
int *nbytes*;

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

cc = **readv(** *d*, *iov*, *iovcnt***)**
int *cc*, *d*;
struct iovec **iov*;
int *iovcnt*;

DESCRIPTION

Read attempts to read *nbytes* of data from the object specified by the descriptor *d* into the buffer pointed to by *buf*. **Readv** performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the **iovec** array: *iov*[**0**], *iov*[**1**], ..., *iov*[*iovcnt–1*].

For **readv**, the **iovec** structure is defined as

```
struct iovec {
        caddr_t         iov_base;
        int     iov_len;
};
```

Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. **Readv** will always completely fill an area before proceeding to the next.

On objects that permit seeking, the **read** starts at a position given by the pointer associated with *d*; see **lseek**(2). Upon return from **read**, the pointer increments by the number of bytes actually read.

Objects that do not permit seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Upon successful completion, **read** and **readv** return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested only if the descriptor refers to a file in which that many bytes remain before the end-of-file.

If the returned value is zero, then the call reached an end-of-file.

RETURN VALUE

A successful call returns the number of bytes actually read. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[EBADF]	*D* is not a valid file descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EWOULDBLOCK]	The file descriptor is marked as non-blocking, and a read would block.

In addition, **readv** may return one of the following errors:

[EINVAL]	*Iovcnt* was less than or equal to zero or greater than 16.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

RELATED INFORMATION

dup(2), open(2), pipe(2), socket(2), socketpair(2)

NAME

readlink – read value of a symbolic link

USAGE

cc = **readlink**(*path*, *buf*, *bufsiz*)
int *cc*;
char **path*, **buf*;
int *bufsiz*;

DESCRIPTION

Readlink places the contents of symbolic link named by *path* into the buffer *buf*, which has size *bufsiz*. The contents of the link are not null-terminated when they are returned.

RETURN VALUE

A successful call returns the number of characters in *buf*. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Readlink will fail and the mode of *path* will be unchanged if:

[EPERM]	The *path* argument contains a byte with the high-order bit set.
[ENOENT]	The pathname is too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[ENXIO]	The named file is not a symbolic link.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
[EINVAL]	The named file is not a symbolic link.
[EFAULT]	*Buf* extends outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

RELATED INFORMATION

stat(2), **lstat**(2), **symlink**(2)

NAME

reboot – reboot system or halt processor

USAGE

#include <sys/reboot.h>

reboot(*howto***)**
int *howto***;**

DESCRIPTION

The **reboot** call is normally invoked in the event of unrecoverable system failures. The *howto* parameter is a mask of options passed to the bootstrap program. The bits of *howto* contain RB_HALT, which causes the processor to halt with no reboot taking place. Currently, the system call interface only permits RB_HALT to be passed to the **reboot** program.

RETURN VALUES

A successful call never returns. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

[EPERM] The caller is not the super-user.

RELATED INFORMATION

halt(8), reboot(8)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

USAGE

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

cc = **recv(** *s*, *buf*, *len*, *flags*)
int *cc*, *s*;
char **buf*;
int *len*, *flags*;

cc = **recvfrom(** *s*, *buf*, *len*, *flags*, *from*, *fromlen*)
int *cc*, *s*;
char **buf*;
int *len*, *flags*;
struct sockaddr **from*;
int **fromlen*;

cc = **recvmsg(** *s*, *msg*, *flags*)
int *cc*, *s*;
struct msghdr *msg*[ ];
int *flags*;

DESCRIPTION

Recv, **recvfrom**, and **recvmsg** receive messages from a socket.

The **recv** call may be used only on a connected socket (see **connect**(2)), while **recvfrom** and **recvmsg** may be used to receive data on a socket whether it is connected or not.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the buffer supplied, excess bytes may be discarded, depending on the type of socket from which the message is received; see **socket**(2).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see **ioctl**(2)), in which case a *cc* of -1 is returned and the external variable *errno* is set to EWOULDBLOCK.

The **select**(2) call may be used to determine whether more data has arrived.

The *flags* argument to a send call comes from the logical OR of one or more of the values,

```
#define MSG_PEEK  0x1   /* peek at incoming message */
#define MSG_OOB   0x2   /* process out-of-band data */
```

The **recvmsg** call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in *<sys/socket.h>*:

```
struct msghdr {
        caddr_t   msg_name;          /* optional address */
        int       msg_namelen;       /* size of address */
        struct    iov *msg_iov;      /* scatter/gather array */
        int       msg_iovlen;        /* # elements in msg_iov */
        caddr_t   msg_accrights;     /* access rights sent/received */
        int       msg_accrightslen;
};
```

Here **msg_name** and **msg_namelen** specify the destination address if the socket is unconnected; **msg_name** may be given as a null pointer if no names are desired or required. The **msg_iov** and **msg_iovlen** describe the scatter/gather locations, as described in **read**(2). Access rights to be sent along with the message are specified in **msg_accrights**, which has length **msg_accrightslen**.

RETURN VALUE

A successful call returns the number of bytes received. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The calls fail if:

[EBADF] The argument *s* is an invalid descriptor.

[ENOTSOCK] The argument *s* is not a socket.

[EWOULDBLOCK] The socket is marked non-blocking and the receive operation would block.

[EINTR] The receive was interrupted by delivery of a signal before any data was available for the receive.

[EFAULT] The call specified that data was to be received into a non-existent or protected part of the process address space.

RELATED INFORMATION

read(2), send(2), socket(2)

NAME

rename – change the name of a file

USAGE

rename(*from*, *to*)
char **from*, **to*;

DESCRIPTION

Rename causes the link named *from* to be renamed with name *to*. If a file named *to* existed before the call to **rename**, it is removed. Both *from* and *to* must be objects of the same type (that is, both directories or both non-directories), and both must reside on the same file system.

Rename guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Rename will fail and neither of the argument files will be affected if any of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *from* does not exist.
[EPERM]	The file named by *from* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (i.e., file systems). Note that this error code will not be returned if the implementation permits cross-device links.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[EINVAL]	*From* is a parent directory of *to*.

RELATED INFORMATION

open(2)

NAME

rmdir – remove a directory file

USAGE

rmdir(*path***)**
char ***path*;**

DESCRIPTION

Rmdir removes the directory file named by *path*. The directory must be empty (a directory that only contains the entries "." and ".." is considered to be empty).

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[ENOTEMPTY]	The named directory is not empty.
[EPERM]	The pathname contains a character with the high-order bit set.
[ENOENT]	The pathname is too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed resides on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

RELATED INFORMATION

mkdir(2), unlink(2)

NAME

select – synchronous I/O multiplexing

USAGE

#include <sys/time.h>

nfound = select(*nfds*, *readfds*, *writefds*, *execptfds*, *timeout*)
int *nfound*, *nfds*, **readfds*, **writefds*, **execptfds*;
struct timeval **timeout*;

DESCRIPTION

Select examines the I/O descriptors specified by the bit masks *readfds*, *writefds*, and *execptfds* to see if they are ready for reading, writing, or if they have an exception condition pending, respectively. The bit "1<<f" in the mask represents the file descriptor *f*. *Nfds* descriptors are checked, i.e., the function examines the bits from zero through *nfds*-1 in the masks. **Select** returns, in place, a mask of those descriptors that are ready. The total number of ready descriptors is returned in *nfound*.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, **select** blocks indefinitely. To poll all of the I/O descriptors without waiting, the *timeout* argument should be non-zero, and should point to a zero-valued **timeval** structure.

Any of *readfds*, *writefds*, and *execptfds* may be set to zero where these descriptors are not of interest.

NOTES

The descriptor masks are always modified on return, even if the call returns as the result of the time-out.

RETURN VALUE

Select returns the number of descriptors that are contained in the bit masks, or –1 if an error occurred. If the time limit expires, then **select** returns zero.

ERRORS

An error return from **select** indicates:

[EBADF] One of the bit masks specified an invalid descriptor.

[EINTR] A signal was delivered before any of the selected events occurred or the time limit expired.

RELATED INFORMATION

accept(2), connect(2), getitimer(2), read(2), write(2), recv(2), send(2)

NAME

send, sendto, sendmsg – send a message from a socket

USAGE

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

cc = **send(** *s*, *msg*, *len*, *flags*)
int *cc*, *s*;
char **msg*;
int *len*, *flags*;

cc = **sendto(** *s*, *msg*, *len*, *flags*, *to*, *tolen*)
int *cc*, *s*;
char **msg*;
int *len*, *flags*;
struct sockaddr **to*;
int *tolen*;

cc = **sendmsg(** *s*, *msg*, *flags*)
int *cc*, *s*;
struct msghdr *msg*[ ];
int *flags*;

DESCRIPTION

Send, **sendto**, and **sendmsg** transmit messages to another socket. **Send** can be used only when the socket is connected, while **sendto** and **sendmsg** can be used at any time.

The address of the target is given by *to*, and *tolen* specifies its size. The length of the message is given by *len*. If the message is too long to pass through the underlying protocol, the error EMSGSIZE is returned and the message is not transmitted. The value -1 may be returned for some locally-detected errors.

If no message space is available at the socket to hold the message to be transmitted, **send** normally blocks, unless the socket has been placed in non-blocking I/O mode. The **select**(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may be set to MSG_OOB to send out-of-band data on sockets that support this form (e.g., SOCK_STREAM).

See **recv(2)** for a description of the *msghdr* structure.

RETURN VALUE

A successful call returns the number of characters sent. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent in one piece. The size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.

RELATED INFORMATION

recv(2), socket(2)

NAME

set_sbrk_size – define memory available for allocation (obsolete)

USAGE

set_sbrk_size (*newsize*)
int *newsize*;

DESCRIPTION

The DOMAIN/IX SR9.0 function **set_sbrk_size**, which defined the amount of memory available for allocation by the memory allocation functions **sbrk**(2), **brk**(2), **malloc**(3), **realloc**(3), and **calloc**(3), is obsolete.

The amount of memory available to these functions is now limited only by the amount of virtual address space available to the process. Any **set_sbrk_size** call that may be in the program is ignored.

We include set_sbrk_size here for backward compatability. However, we do not encourage its continued use, and we cannot promise its continued support.

RELATED INFORMATION

brk(2), **sbrk**(2), **calloc**(3), **malloc**(3), **realloc**(3) **environ**(7)

NAME

set_version, **get_version** – set/get system version (obsolete)

USAGE

set_version(*string***)**
char **string*;

get_version (*cp***)**
char *cp***[16];**

DESCRIPTION

These calls are obsolete. We include them in this release for compatability only. However, we do not encourage their continued use, and we cannot promise their continued support.

The DOMAIN/IX function **set_version** allows programs to specify the version of DOMAIN/IX — AT&T UNIX System V or Berkeley 4.2 UNIX — that will be used to define arguments and semantics for certain system and library functions. Valid **string** arguments are "bell" and "berkeley". The default version is "bell". The selected version is inherited across program invocation, exec(2), and by forked children.

The DOMAIN/IX function **get_version** returns a string identifying the version of UNIX (Bell UNIX System V or Berkeley UNIX) currently interpreting arguments and semantics for certain system and library functions. It returns either "bell" or "berkeley".

RELATED INFORMATION

getpgrp(2), **setpgrp**(2)

NAME

setgroups – set group access list

USAGE

#include <sys/param.h>

setgroups(*ngroups*, *gidset*)
int *ngroups*, **gidset*;

DESCRIPTION

Setgroups sets the group access list of the current user process to the one specified by the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in *<sys/param.h>*.

Only the super-user can set new groups.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The setgroups call fails if:

[EPERM] The caller is not the super-user.

[EFAULT] The address specified for *gidset* is outside the process's legal address space.

RELATED INFORMATION

getgroups(2), initgroups(3X)

NAME

setpgrp – set process group

USAGE

setpgrp(*pid*, *pgrp***)**
int *pid*, *pgrp*;

DESCRIPTION

Setpgrp sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process.

If the caller is not the super-user, then the affected process must have the same effective user-ID as the caller, or must be a descendant of the calling process.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Setpgrp fails and the process group is not altered if any of the following occurs:

[ESRCH] The requested process does not exist.

[EPERM] The effective user ID of the requested process is different from that of the caller, and the process is not a descendant of the calling process.

RELATED INFORMATION

getpgrp(2)

NAME

setregid – set real and effective group ID

USAGE

setregid(*rgid*, *egid*)
int *rgid*, *egid*;

DESCRIPTION

For the current process, **setregid** sets the real group ID to *rgid* and the effective group ID to *egid*. Only the super-user may change the real group ID of a process. Other users may only change the effective group ID to the real group ID.

If you supply a value of -1 for either *rgid* or *egid*, the system substitutes the current ID in place of the -1 parameter.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective group-ID to the real group-ID was specified.

RELATED INFORMATION

getgid(2), setreuid(2), setgid(3)

NAME

setreuid – set real and effective user ID

USAGE

setreuid(*ruid*, *euid*)
int *ruid*, *euid*;

DESCRIPTION

For the current process, **setreuid** sets the real user ID to *ruid* and the effective user ID to *euid*. Only the super-user may change the real user ID of a process. Other users may only change the effective user ID to the real user ID.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective group-ID to the real group-ID was specified.

RELATED INFORMATION

getuid(2), setregid(2), setuid(3)

NAME

shutdown – shut down part of a full-duplex socket connection

USAGE

shutdown(*s*, *how*)
int *s*, *how*;

DESCRIPTION

The **shutdown** call closes down all or part of a full-duplex connection on the socket associated with *s*. The *how* parameter may be any of:

0	no further receives are allowed.
1	no further sends are allowed.
2	no further sends or receives are allowed.

DIAGNOSTICS

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The call succeeds unless:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

RELATED INFORMATION

connect(2), socket(2)

NAME

sigblock – block signals

USAGE

sigblock(*mask***);**
int *mask***;**

DESCRIPTION

Sigblock adds the signals specified in *mask* to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*th bit in **mask** is a 1.

You cannot block SIGKILL, SIGSTOP, or SIGCONT.

RETURN VALUE

The previous set of masked signals is returned.

RELATED INFORMATION

kill(2), sigvec(2), sigsetmask(2),

NAME

sigpause – atomically release blocked signals and wait for interrupt

USAGE

sigpause(*sigmask***)**
int *sigmask***;**

DESCRIPTION

Sigpause assigns *sigmask* to the set of masked signals, then waits for a signal to arrive. On return, the set of masked signals is restored. *Sigmask* is usually set to zero to indicate that no signals should be blocked. **Sigpause** always terminates by being interrupted, and always returns EINTR.

In normal usage, a signal may be blocked using **sigblock**(2); to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses by using **sigpause** with the mask returned by **sigblock.**

RETURN VALUE

Sigpause returns EINTR.

RELATED INFORMATION

sigblock(2), **sigvec**(2)

NAME
sigsetmask – set current signal mask

USAGE
sigsetmask(*mask***);**
int *mask***;**

DESCRIPTION
Sigsetmask sets the current signal mask (those signals that are blocked from delivery). Signal *i* is blocked if the *i*th bit in *mask* is a 1.

You cannot block SIGKILL, SIGSTOP, or SIGCONT.

RETURN VALUE
The previous set of masked signals is returned.

RELATED INFORMATION
kill(2), sigvec(2), sigblock(2), sigpause(2)

NAME

sigstack – set and/or get signal stack context

USAGE

```
#include &lt;signal.h&gt;

struct sigstack {
        caddr_t     ss_sp;
        int         ss_onstack;
};

sigstack(ss, oss);
struct sigstack *ss, *oss;
```

DESCRIPTION

Sigstack allows you to define an alternate stack on which to process signals. The DOMAIN/IX implementation of **sigstack** is a no-op, included for compatability with existing programs.

If *ss* is non-zero, it specifies a "signal stack" on which to deliver signals and tells the system whether the process is currently executing on that stack.

NOTES

DOMAIN/IX does not implement a signal stack. Calls to **sigstack** always return 0, and the stack context is never changed.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Sigstack will fail and the signal stack context will remain unchanged if the following occurs:

[EFAULT] Either *ss* or *oss* points to memory that is not a valid part of the process's address space.

RELATED INFORMATION

sigvec(2), setjmp(3)

NAME

sigvec – software signal facilities

USAGE

```
#include <signal.h>
struct sigvec {
        int     (*sv_handler)();
        int     sv_mask;
        int     sv_onstack;
};

sigvec( sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked, the current process context is saved, and a new one is built. A signal may be blocked, ignored, or delivered to a handler, as the process requires. A process may also specify a default action for the system to take when a given signal occurs. Normally, signal handlers execute on the current stack of the process.

All signals have the same priority. While a signal routine executes, the signal that triggered it is blocked, although other signals may occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally zero). It may be changed with a **sigblock**(2) or **sigsetmask**(2) call, and when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process, then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that, if the signal handling routine returns, the process will normally resume execution in the state it was in before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler (or until a **sigblock** or **sigsetmask** call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and including, with a logical OR, the signal mask associated with the handler to be invoked.

Sigvec assigns a handler for a specific signal. If *vec* is non-zero, it specifies a handler routine and mask to be used when delivering the specified signal. Further, if **sv_onstack** is 1, some systems will deliver the signal to the process on a signal stack, as specified with **sigstack**(2). (This feature is not implemented in DOMAIN/IX.) If *ovec* is non-zero, the previous handling information for the signal is returned to the user.

The following is a list of all signals with names as in the include file <*signal.h*>:

SIGHUP	1	hang-up
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction
SIGTRAP	5	trace trap
SIGIOT	6	IOT instruction
SIGEMT	7	EMT instruction
SIGFPE	8	floating-point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	death of a child
SIGAPOLLO	19	DOMAIN System fault with no UNIX System equivalent
SIGSTOP	20†	stop, cannot be caught, held, or ignored
SIGTSTP	21†	stop signal generated from keyboard
SIGCONT	22•	continue after stop
SIGCHLD	23•	child status has changed
SIGTTIN	24†	background read attempted from control terminal
SIGTTOU	25†	background write attempted to control terminal
SIGIO	26	I/O is possible on a descriptor
SIGTINT	26	input record is available at control terminal
SIGXCPU	27	cpu time limit exceeded
SIGXFSZ	28	file size limit exceeded
SIGVTALRM	29	virtual time alarm
SIGPROF	30	profiling timer alarm
SIGURG	31•	urgent condition present on socket

Once a signal handler is installed, it remains installed until another **sigvec** call is made, or an **execve**(2) is performed. The default action for a signal may be reinstated by setting **sv_handler** to SIG_DFL; this default is termination except for signals marked with a bullet (•) or a dagger (†). Signals marked with a bullet are discarded if the action is SIG_DFL; signals marked with a dagger cause the process to stop. If **sv_handler** is SIG_IGN, the signal is subsequently ignored, and pending instances of the signal are discarded.

If a caught signal occurs during certain system calls and causes the call to terminate prematurely, the call is automatically restarted. This is especially likely to occur during a **read**(2) or **write**(2) on a slow device (e.g., a terminal) and during a **wait**(2).

After a **fork**(2) or **vfork**(2), the child inherits all signals, the signal mask, and the signal stack.

Execve(2) resets all caught signals to default action; ignored signals remain ignored; the signal mask remains the same; and the signal stack state is reset.

NOTES

The signal stack feature is not implemented on DOMAIN Systems. Calls to **sigstack**(2) always return 0. Stack context is not changed.

DOMAIN systems send the signal SIGAPOLLO whenever a fault occurs that is not otherwise mapped into a signal. Typical generators of SIGAPOLLO include network failures, display-acquire timeouts, and disk full errors.

The system does not allow the mask specified in *vec* to block SIGKILL, SIGSTOP, or SIGCONT.

The handler routine can be declared as follows:

> *handler*(*sig*, *code*, *scp*)
> **int** *sig*, *code*;
> **struct sigcontext** **scp*;

Here, *sig* is the signal number into which the hardware faults and traps are mapped as defined below. *Code* is a 32-bit value. If the signal is SIGAPOLLO, *code* is the DOMAIN System status code describing the fault. (To generate a list of DOMAIN System status codes and brief explanations of their meanings, run the command /systest/ssr_util/all_stcode.) Otherwise, *code* is a value associated with one of the constants listed below. *Scp* is a pointer to the **sigcontext** structure (defined in <**signal.h**>), which is used to restore the context from before the signal.

DOMAIN System Hardware traps are mapped to signals and codes as indicated below. All of these symbols are defined in <*signal.h*>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	
Protection violation	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_PRIVIN_FAULT
Reserved addressing	SIGILL	ILL_RESOP_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Sigvec will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT] Either *vec* or *ovec* points to memory that is not a valid part of the process's address space.

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default, SIGCONT is ignored).

RELATED INFORMATION

kill(1), kill(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), sigvec(2), setjmp(3), tty(4)

NAME

socket – create an endpoint for communication

USAGE

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

s = **socket(** *af*, *type*, *protocol*)
int *s*, *af*, *type*, *protocol*;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies the address format according to which addresses specified by later operations at the socket should be interpreted. These formats are defined in the include file <**sys/socket.h**>. The only format currently available is:

AF_INET (ARPA Internet addresses),

The socket has the indicated *type*, which specifies the semantics of communication. Possible *type*s are:

SOCK_STREAM
SOCK_DGRAM

Type SOCK_STREAM provides sequenced, reliable, two-way connection-based byte streams with an out-of-band data transmission mechanism. Type SOCK_DGRAM supports datagrams (i.e., connectionless, unreliable messages of a fixed (typically small) maximum length).

The *protocol* specifies a particular protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address format. However, many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication environment" in which communication is to take place; see **services**(5) and **protocols**(5).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be connected before any data can be sent or received on it. A connection to another socket is created with a **connect**(2) call. Once connected, data may be transferred using **read**(2) and **write**(2) calls or some variant of the **send**(2) and **recv**(2) calls. When a session is over, a **close**(2) is performed. Out-of-band data may also be transmitted as described in **send**(2) and received as described in **recv**(2).

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and calls will indicate an error with a return of -1 and with ETIMEDOUT as the specific code in the global variable *errno*. The protocols may keep sockets active by forcing transmissions roughly every minute in the absence of other activity. An error is indicated if no response can be elicited on an otherwise idle connection for a extended time period (e.g., 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes processes that do not handle the signal to exit.

SOCK_DGRAM sockets allow the sending of datagrams to correspondents named in **send**(2) calls. You may receive datagrams at such a socket with **recv**(2).

An **fcntl**(2) call can be used to specify a process group that will receive a SIGURG signal when the out-of-band data arrives.

RETURN VALUE

A successful call returns a descriptor referencing the socket. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The socket call fails if:

[EAFNOSUPPORT]	The specified address family is not supported in this version of the system.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[EMFILE]	The per-process descriptor table is full.
[ENOBUFS]	No buffer space is available. The socket cannot be created.

RELATED INFORMATION

accept(2), **bind**(2), **connect**(2), **getsockname**(2), **getsockopt**(2), **ioctl**(2), **listen**(2), **recv**(2), **select**(2), **send**(2), **shutdown**(2), **socketpair**(2)

NAME

socketpair – create a pair of connected sockets

USAGE

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

socketpair(*d*, *type*, *protocol*, *sv*)
int *d*, *type*, *protocol*;
int *sv*[2];

DESCRIPTION

The **socketpair** call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[**0**] and *sv*[**1**]. The two sockets are indistinguishable.

DIAGNOSTICS

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

RELATED INFORMATION

read(2), write(2), pipe(2)

NAME

soft_link, **soft_unlink** – create or delete soft links

USAGE

int soft_link(*linktext*, *pathname*)
char **linktext*, **pathname*;

int soft_unlink(*pathname*)
char **pathname*;

DESCRIPTION

The DOMAIN/IX system call **soft_link** creates a "soft" link to a specified file. On DOMAIN systems, a soft link contains "link text" that references the pathname of an object. A "hard" link to an object is, in most cases, indistinguishable from the object itself.

The *pathname* argument is the pathname of the link to be created or deleted. The *linktext* argument is the pathname of the file to which the link points. The file named by *linktext* need not exist.

The system call **soft_unlink** deletes a soft link, leaving the object to which the link points intact. To delete a hard link, use **unlink**(2).

DIAGNOSTICS

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

RELATED INFORMATION

link(2), **symlink**(2), **unlink**(2)

NAME

stat, lstat, fstat – get file status

USAGE

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

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

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

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

DESCRIPTION

Stat obtains information about the file *path*. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be traversable.

Lstat is like **stat**, except in the case where the named file is a symbolic link. In this case, **lstat** returns information about the link, while **stat** returns information about the file to which the link refers.

Fstat obtains the same information about the open file to which *fd* refers (similar to the information returned by an **open** call).

In all cases, *buf* is a pointer to a **stat** structure into which information about the file is placed. The contents of this structure are:

```
struct stat {
        dev_t     st_dev;     /* device inode resides on */
        ino_t     st_ino;     /* this inode's number */
        u_short   st_mode;    /* protection */
        short     st_nlink;   /* number or hard links to the file */
        short     st_uid;     /* user-id of owner */
        short     st_gid;     /* group-id of owner */
        dev_t     st_rdev;    /* the device type, for inode that is device */
        off_t     st_size;    /* total size of file */
        time_t    st_atime;   /* file last access time */
```

```
        int        st_spare1;
        time_t     st_mtime;   /* file last modify time */
        int        st_spare2;
        time_t     st_ctime;   /* file last status change time */
        int        st_spare3;
        long       st_blksize; /* optimal blocksize for file system i/o ops */
        long       st_blocks;  /* actual number of blocks allocated */
        long       st_spare4[2];
};
```

st_atime Time when file data was last read or modified. Changed by the following system calls: **mknod**(2), **utimes**(2), **read**(2), and **write**(2). For reasons of efficiency, **st_atime** is not set when a directory is searched.

st_mtime Time when data was last modified. It is not set by changes of owner, group, link count, or mode. Changed by the following system calls: **mknod**(2), **utimes**(2), **write**(2).

st_ctime Time when file status was last changed. It is set both both by writing and changing the i-node. Changed by the following system calls: **chmod**(2) **chown**(2), **link**(2), **mknod**(2), **unlink**(2), **utimes**(2), **write**(2).

The status information word **st_mode** has bits:

```
#define S_IFMT     0170000   /* type of file */
#define S_IFDIR    0040000   /* directory */
#define S_IFCHR    0020000   /* character special */
#define S_IFBLK    0060000   /* block special */
#define S_IFREG    0100000   /* regular */
#define S_IFLNK    0120000   /* symbolic link */
#define S_IFSOCK   0140000   /* socket */
#define S_ISUID    0004000   /* set user id on execution */
#define S_ISGID    0002000   /* set group id on execution */
#define S_ISVTX    0001000   /* save swapped text even after use */
#define S_IREAD    0000400   /* read permission, owner */
#define S_IWRITE   0000200   /* write permission, owner */
#define S_IEXEC    0000100   /* execute/search permission, owner */
```

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

When *fd* is associated with a pipe, **fstat** reports an ordinary file with an inode number, restricted permissions, and a length (that may not be correct).

NOTES

Applying **fstat** to a socket returns a zeroed buffer.

The fields in the stat structure currently marked **st_spare1**, **st_spare2**, and **st_spare3** are intended to allow future expansion of inode time stamps to 64 bits. Their existence may cause problems for programs that depend on the time stamps being contiguous (in calls to **utimes**(2)).

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

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

[EPERM] The pathname contains a character with the high-order bit set.

[ENOENT] The pathname is too long.

[ENOENT] The named file does not exist.

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

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

[ELOOP] The call encountered too many symbolic links in translating the pathname.

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

[EBADF] *Fd* is not a valid open file descriptor.

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

RELATED INFORMATION

chmod(2), **chown**(2), **utimes**(2)

NAME

symlink – make symbolic link to a file

USAGE

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

DESCRIPTION

Symlink creates a symbolic link named *name2* that references the object named by *name1* (*name2* is the name of the file created, and *name1* is the string used in creating the symbolic link). Either name may be an arbitrary pathname; the files need not be on the same file system.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

The symbolic link is made unless on or more of the following are true:

[EPERM]	Either *name1* or *name2* contains a character with the high-order bit set.
[ENOENT]	One of the pathnames specified is too long.
[ENOTDIR]	A component of the *name2* prefix is not a directory.
[EEXIST]	*Name2* already exists.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EROFS]	The file *name2* would reside on a read-only file system.
[EFAULT]	Either *name1* or *name2* points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

RELATED INFORMATION

link(2), **unlink**(2)

NAME

sync – update super-block

USAGE

void sync()

DESCRIPTION

The **sync** system call force writes information in memory to disk.

The **sync** operation is not actually necessary on DOMAIN hardware, because the system buffers are automatically written to disk at shutdown. We provide it in the interest of ensuring compatibility with other implementations.

RELATED INFORMATION

fsync(2), **sync**(8), **update**(8)

NAME

truncate – truncate a file to a specified length

USAGE

truncate(*path*, *length*)
char **path*;
int *length*;

ftruncate(*fd*, *length*)
int *fd*, *length*;

DESCRIPTION

Truncate truncates the file named by *path* to a maximum of *length* bytes in size. **Ftruncate** does the same thing for the file referenced by *fd*, which must be open for writing.

If the file was larger than *length*, the extra data is lost.

NOTES

Partial blocks discarded as the result of truncation are not zero-filled; this can leave holes in files which do not read as zero.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

Truncate succeeds unless:

[EPERM]	The pathname contains a character with the high-order bit set.
[ENOENT]	The pathname is too long.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the *path* prefix denies search permission.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (i.e., shared text) file that is being executed.
[EFAULT]	*Path* points outside the process's allocated address space.

Ftruncate succeeds unless:

[EBADF] *Fd* is not a valid descriptor.

[EINVAL] *Fd* refers to a socket, not a file.

RELATED INFORMATION

open(2)

NAME

umask – set/get file creation mask

USAGE

int umask(*cmask*)
int *cmask*;

DESCRIPTION

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

RETURN VALUE

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

RELATED INFORMATION

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

NAME

unlink – remove directory entry

USAGE

unlink(*path*)
char **path*;

DESCRIPTION

Unlink removes the entry for the file *path* from its directory. If this entry was the last link to the file and no process has the file open, the system reclaims all resources associated with the file. If a process has the file open, the system waits until the file is closed before reclaiming resources, even though the directory entry has disappeared.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno*.

ERRORS

The **unlink** succeeds unless:

[EPERM]	The path contains a character with the high-order bit set.
[ENOENT]	The pathname is too long.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The named file is a directory and the effective user ID of the process is not the super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

RELATED INFORMATION

close(2), link(2), rmdir(2)

NAME

utimes – set file times

USAGE

#include <sys/times.h>

utimes(*file*, *tv*)
char **file*;
struct timeval *tv*[2];

DESCRIPTION

The **utimes** call uses the "accessed" and "updated" times in that order from the *tv* vector to set the corresponding recorded times for *file*.

The caller must be the owner of the file or the super-user. The "inode-changed" time of the file is set to the current time.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno* as indicated below.

ERRORS

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

[EPERM]	The pathname contains a character with the high-order bit set.
[ENOENT]	The pathname is too long.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	A component of the path prefix denies search permission.
[EPERM]	The process is not super-user and not the owner of the file.
[EACCES]	The effective user ID of the caller is not super-user or the owner of the file.
[EROFS]	The file system containing the file is mounted read-only.
[EFAULT]	*Tv* points outside the process's allocated address space.
[ELOOP]	The call encountered too many symbolic links in translating the pathname.

RELATED INFORMATION

stat(2)

NAME

vfork – spawn a new process in a more efficient way

USAGE

pid = **vfork()**
int *pid*;

DESCRIPTION

Vfork creates new processes without fully copying the address space of the old process. This conserves resources in a paged environment. **Vfork** is primarily useful when the purpose of **fork**(2) is to create a new system context for an **execve**(2). **Vfork** differs from **fork** in that the child borrows the parent's memory and thread of control until a call to **execve** or an exit (either by a call to **exit**(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns zero in the child's context and (later) the PID of the child in the parent's context.

Vfork can normally be used just like **fork.** However, it is illegal to return from the procedure that called **vfork** while running in the child process, since by so doing, **vfork** would be attempting to return to a non-existent stack frame. Be careful, also, to call **_exit** rather than **exit** if you can't **execve**, since **exit** will flush and close standard I/O channels, and thereby affect the parent process's standard I/O data structures. (Even with **fork**, it is better not to call **exit** since buffered data is then flushed twice.)

NOTES

In a future release, this system call may be eliminated in favor of a more effective process creation mechanism.

To avoid possible deadlocks, processes that are children in the middle of a **vfork** are never sent SIGTTOU or SIGTTIN signals; rather, output or **ioctls** are allowed, and input attempts result in an end-of-file indication.

RETURN VALUE

Upon successful completion, **vfork** returns zero to the child process and returns the child's process ID to the parent process. Otherwise, -1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

ERRORS

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

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

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

RELATED INFORMATION

fork(2), execve(2), sigvec(2), wait(2),

NAME

wait, wait3 – wait for process to terminate

USAGE

#include <sys/wait.h>

pid = **wait(** *status* **)**
int *pid*;
union wait **status*;

pid = **wait(0)**
int *pid*;

#include <sys/time.h>
#include <sys/resource.h>

pid = **wait3(** *status*, *options*, *rusage* **)**
int *pid*;
union wait **status*;
int *options*;
struct rusage **rusage*;

DESCRIPTION

Wait forces its caller to delay until a signal is received or until one of its child processes terminates. If any child process has died since the last **wait**, **wait** returns immediately and gives the process ID and exit status of one of the terminated children. If there are no children, the caller also returns immediately with the value -1.

Upon return from a successful **wait** call, *status* is nonzero, and the high byte of *status* contains the low byte of the argument to **exit** supplied by the child process; the low byte of *status* contains the termination status of the process. A more precise definition of the *status* word is given in *<sys/wait.h>*.

Wait3 provides an alternate interface for programs that must not block when collecting the status of child processes. The *status* parameter is defined as above. The *options* parameter is one of

WNOHANG the call should not block if there are no processes that wish to report status.

WUNTRACED only children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal should have their status reported.

If **rusage** is non-zero, a summary of the resources used by the terminated process and all its children is returned (this information is currently not available for stopped processes).

When the WNOHANG option is specified and no processes wish to report status, **wait3** returns a *PID* of zero. The WNOHANG and WUNTRACED options may be combined by OR'ing the two values.

NOTES

See **sigvec(2)** for a list of termination statuses (signals); zero status indicates normal termination. A special status (0177) is returned for a stopped process that has not terminated and can be restarted.

If the parent process terminates without waiting on its children, the children become orphans. On DOMAIN Systems, the parent process ID of all orphan processes is set to that of the Display Manager (process 1), even though no real parent-child relationship exists between the two (e.g., the DM cannot be made to **wait** on these "children").

Wait and **wait3** are automatically restarted when a process receives a signal while awaiting termination of a child process.

RETURN VALUE

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

Wait3 returns -1 if there are no children not previously waited for. It returns zero if WNOHANG is specified and there are no stopped or exited children.

ERRORS

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

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] The *status* or *rusage* arguments point to an illegal address.

RELATED INFORMATION

exit(2)

NAME

write, writev – write on a file

USAGE

write(*d*, *buf*, *nbytes*)
int *d*;
char **buf*;
int *nbytes*;

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

writev(*d*, *iov*, *ioveclen*)
int *d*;
struct iovec **iov*;
int *ioveclen*;

DESCRIPTION

Write attempts to write *nbytes* of data to the object referred to by the descriptor *d* from the buffer pointed to by *buf*. **Writev** performs the same action, but gathers the output data from the *iovlen* buffers specified by the members of the **iovec** array: *iov*[**0**], *iov*[1], etc.

On objects that allow seeking, the **write** starts at a position given by the pointer associated with *d*; see lseek(2). Upon return from **write**, the pointer is incremented by the number of bytes actually written.

On objects that do not allow seeking, the **write** always occurs at the current position. The value of the pointer associated with such an object is undefined.

NOTES

In DOMAIN/IX, **write** does not clear *setuid*.

RETURN VALUE

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

ERRORS

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

[EBADF] *D* is not a valid descriptor open for writing.

[EPIPE] An attempt was made to write to a pipe that is not open for reading by any process.

[EPIPE] An attempt was made to write to a pipe or socket of type SOCK_STREAM that is not connected to a peer socket.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

[EFAULT] Part of *iov* or data to be written to the file points outside the process's allocated address space.

RELATED INFORMATION

lseek(2), open(2), pipe(2)

This is a topical index for Section 2 of the *DOMAIN/IX Programmer's Reference Manual for BSD4.2*. For a permuted index of all reference information, see Appendix A of this manual.

NAME

intro – introduction to library functions

DESCRIPTION

This section describes functions implemented (on DOMAIN/IX Systems) in the libraries */lib/clib* and */lib/unixlib*. In this section, functions are grouped alphabetically by subsection. The subsections in this section reflect the original UNIX system library structure, under which these routines were distributed across a larger number of libraries.

(3) These are the standard C library functions. (On DOMAIN Systems, *clib* also includes all the functions described in section 2.)

(3M) These functions constitute the math library (included in *clib*). They are automatically loaded as needed. Declarations for these functions may be obtained from the include file **<math.h>**.

(3N) These functions constitute the internet network library (included in *clib*)

(3S) These functions constitute the "standard I/O package", see **intro**(3S). Declarations for these functions may be obtained from the include file **<stdio.h>**.

(3X) These are miscellaneous functions.

(3C) Routines included for compatibility with other systems. In particular, a number of system call interfaces provided in previous releases of DOMAIN/IX have been included for source code compatibility. The manual entry for each compatibility routine indicates the proper interface to use.

DIAGNOSTICS

Math functions (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 (domain error) or ERANGE (range error). The values of EDOM and ERANGE are defined in the include file **<math.h>**.

FILES

/lib/clib	The C language library
/lib/unixlib	UNIX System calls.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
abort	abort.3	generate a fault
abs	abs.3	integer absolute value
acos	sin.3m	trigonometric functions
alarm	alarm.3c	schedule signal after specified time

asctime	ctime.3	convert date and time to ASCII
asin	sin.3m	trigonometric functions
assert	assert.3x	program verification
atan	sin.3m	trigonometric functions
atan2	sin.3m	trigonometric functions
atof	atof.3	convert ASCII to numbers
atoi	atof.3	convert ASCII to numbers
atol	atof.3	convert ASCII to numbers
cabs	hypot.3m	Euclidean distance
calloc	malloc.3	memory allocator
ceil	floor.3m	absolute value, floor, ceiling functions
clearerr	ferror.3s	stream status inquiries
closedir	directory.3	directory operations
cos	sin.3m	trigonometric functions
cosh	sinh.3m	hyperbolic functions
ctime	ctime.3	convert date and time to ASCII
curses	curses.3x	screen functions with optimal cursor motion
dbminit	dbm.3x	database subroutines
delete	dbm.3x	database subroutines
ecvt	ecvt.3	output conversion
edata	end.3	last locations in program
end	end.3	last locations in program
endgrent	getgrent.3	get group file entry
endhostent	gethostent.3n	get network host entry
endnetent	getnetent.3n	get network entry
endprotoent	getprotoent.3n	get protocol entry
endpwent	getpwent.3	get password file entry
endservent	getservent.3n	get service entry
environ	execl.3	execute a file
etext	end.3	last locations in program
exec	execl.3	execute a file
exece	execl.3	execute a file
execl	execl.3	execute a file
execle	execl.3	execute a file
execlp	execl.3	execute a file
exect	execl.3	execute a file
execv	execl.3	execute a file
execvp	execl.3	execute a file
exit	exit.3	terminate a process after flushing any pending output
exp	exp.3m	exponential, logarithm, power, square root
fabs	floor.3m	absolute value, floor, ceiling functions
fclose	fclose.3s	close or flush a stream

fcvt	ecvt.3	output conversion
feof	ferror.3s	stream status inquiries
ferror	ferror.3s	stream status inquiries
fetch	dbm.3x	database subroutines
fflush	fclose.3s	close or flush a stream
fgetc	getc.3s	get character or word from stream
fgets	gets.3s	get a string from a stream
fileno	ferror.3s	stream status inquiries
firstkey	dbm.3x	database subroutines
floor	floor.3m	absolute value, floor, ceiling functions
fprintf	printf.3s	formatted output conversion
fputc	putc.3s	put character or word on a stream
fputs	puts.3s	put a string on a stream
fread	fread.3s	buffered binary input/output
free	malloc.3	memory allocator
frexp	frexp.3	split into mantissa and exponent
fscanf	scanf.3s	formatted input conversion
fseek	fseek.3s	reposition a stream
ftell	fseek.3s	reposition a stream
ftime	time.3c	get date and time
fwrite	fread.3s	buffered binary input/output
gamma	gamma.3m	log gamma function
gcvt	ecvt.3	output conversion
getc	getc.3s	get character or word from stream
getchar	getc.3s	get character or word from stream
getenv	getenv.3	value for environment name
getgrent	getgrent.3	get group file entry
getgrgid	getgrent.3	get group file entry
getgrnam	getgrent.3	get group file entry
gethostbyaddr	gethostent.3n	get network host entry
gethostbyname	gethostent.3n	get network host entry
gethostent	gethostent.3n	get network host entry
getlogin	getlogin.3	get login name
getnetbyaddr	getnetent.3n	get network entry
getnetbyname	getnetent.3n	get network entry
getnetent	getnetent.3n	get network entry
getpass	getpass.3	read a password
getprotobyname	getprotoent.3n	get protocol entry
getprotobynumber	getprotoent.3n	get protocol entry
getprotoent	getprotoent.3n	get protocol entry
getpw	getpw.3c	get name from uid
getpwent	getpwent.3	get password file entry
getpwnam	getpwent.3	get password file entry
getpwuid	getpwent.3	get password file entry

gets	gets.3s	get a string from a stream
getservbyname	getservent.3n	get service entry
getservbyport	getservent.3n	get service entry
getservent	getservent.3n	get service entry
getw	getc.3s	get character or word from stream
getwd	getwd.3	get current working directory pathname
gmtime	ctime.3	convert date and time to ASCII
gtty	stty.3c	set and get terminal state (defunct)
htonl	byteorder.3n	convert values between host and network byte order
htons	byteorder.3n	convert values between host and network byte order
hypot	hypot.3m	Euclidean distance
index	string.3	string operations
inet_addr	inet.3n	Internet address manipulation routines
inet_lnaof	inet.3n	Internet address manipulation routines
inet_makeaddr	inet.3n	Internet address manipulation routines
inet_netof	inet.3n	Internet address manipulation routines
inet_network	inet.3n	Internet address manipulation routines
initgroups	initgroups.3x	initialize group access list
initstate	random.3	better random number generator
insque	insque.3	insert/remove element from a queue
isalnum	ctype.3	character classification macros
isalpha	ctype.3	character classification macros
isascii	ctype.3	character classification macros
isatty	ttyname.3	find name of a terminal
iscntrl	ctype.3	character classification macros
isdigit	ctype.3	character classification macros
islower	ctype.3	character classification macros
isprint	ctype.3	character classification macros
ispunct	ctype.3	character classification macros
isspace	ctype.3	character classification macros
isupper	ctype.3	character classification macros
ldexp	frexp.3	split into mantissa and exponent
localtime	ctime.3	convert date and time to ASCII
log	exp.3m	exponential, logarithm, power, square root
log10	exp.3m	exponential, logarithm, power, square root
longjmp	setjmp.3	non-local goto
malloc	malloc.3	memory allocator
mktemp	mktemp.3	make a unique file name
modf	frexp.3	split into mantissa and exponent
nextkey	dbm.3x	database subroutines
nice	nice.3c	set program priority
ntohl	byteorder.3n	convert values between host and

		network byte order
ntohs	byteorder.3n	convert values between host and network byte order
opendir	directory.3	directory operations
pause	pause.3c	stop until signal
pclose	popen.3	initiate I/O to/from a process
perror	perror.3	system error messages
popen	popen.3	initiate I/O to/from a process
pow	exp.3m	exponential, logarithm, power, square root
printf	printf.3s	formatted output conversion
psignal	psignal.3	system signal messages
putc	putc.3s	put character or word on a stream
putchar	putc.3s	put character or word on a stream
puts	puts.3s	put a string on a stream
putw	putc.3s	put character or word on a stream
qsort	qsort.3	quicker sort
rand	rand.3c	random number generator
random	random.3	better random number generator
rcmd	rcmd.3x	routines for returning a stream to a remote command
re_comp	regex.3	regular expression handler
re_exec	regex.3	regular expression handler
readdir	directory.3	directory operations
realloc	malloc.3	memory allocator
remque	insque.3	insert/remove element from a queue
rewind	fseek.3s	reposition a stream
rewinddir	directory.3	directory operations
rexec	rexec.3x	return stream to a remote command
rindex	string.3	string operations
rresvport	rcmd.3x	routines for returning a stream to a remote command
ruserok	rcmd.3x	routines for returning a stream to a remote command
scandir	scandir.3	scan a directory
scanf	scanf.3s	formatted input conversion
seekdir	directory.3	directory operations
setbuf	setbuf.3s	assign buffering to a stream
setegid	setuid.3	set user and group ID
seteuid	setuid.3	set user and group ID
setgid	setuid.3	set user and group ID
setgrent	getgrent.3	get group file entry
sethostent	gethostent.3n	get network host entry
setjmp	setjmp.3	non-local goto
setnetent	getnetent.3n	get network entry

setprotoent	getprotoent.3n	get protocol entry
setpwent	getpwent.3	get password file entry
setrgid	setuid.3	set user and group ID
setruid	setuid.3	set user and group ID
setservent	getservent.3n	get service entry
setstate	random.3	better random number generator
setuid	setuid.3	set user and group ID
signal	signal.3c	simplified software signal facilities
sin	sin.3m	trigonometric functions
sinh	sinh.3m	hyperbolic functions
sleep	sleep.3	suspend execution for interval
sprintf	printf.3s	formatted output conversion
sqrt	exp.3m	exponential, logarithm, power, square root
srand	rand.3c	random number generator
srandom	random.3	better random number generator
sscanf	scanf.3s	formatted input conversion
stdio	intro.3s	standard buffered input/output package
store	dbm.3x	database subroutines
strcat	string.3	string operations
strcmp	string.3	string operations
strcpy	string.3	string operations
strlen	string.3	string operations
strncat	string.3	string operations
strncmp	string.3	string operations
strncpy	string.3	string operations
stty	stty.3c	set and get terminal state (defunct)
swab	swab.3	swap bytes
sys_errlist	perror.3	system error messages
sys_nerr	perror.3	system error messages
sys_siglist	psignal.3	system signal messages
system	system.3	issue a Shell command
tan	sin.3m	trigonometric functions
tanh	sinh.3m	hyperbolic functions
telldir	directory.3	directory operations
tgetent	termcap.3x	terminal independent operation routines
tgetflag	termcap.3x	terminal independent operation routines
tgetnum	termcap.3x	terminal independent operation routines
tgetstr	termcap.3x	terminal independent operation routines
tgoto	termcap.3x	terminal independent operation routines
time	time.3c	get date and time
times	times.3c	get process times
timezone	ctime.3	convert date and time to ASCII
tputs	termcap.3x	terminal independent operation routines
ttyname	ttyname.3	find name of a terminal

ungetc	ungetc.3s	push character back into input stream
utime	utime.3c	set file times
valloc	valloc.3	aligned memory allocator
varargs	varargs.3	variable argument list

RELATED INFORMATION

intro(3C), intro(3S), intro(3M), intro(3N), nm(1), ld(1), cc(1), intro(2)

NAME
abort – generate a fault

USAGE
abort()

DESCRIPTION
Abort executes an instruction that is illegal in user mode. This sends a signal that terminates the process. You may examine the remains of the aborted process using the **/com/tb** command.

NOTES
The **abort** function does not flush standard I/O buffers. Use **fflush**(3S) to accomplish this.

DIAGNOSTICS
Usually "IOT trap" from the shell.

RELATED INFORMATION
sigvec(2), exit(2)

NAME

abs – integer absolute value

USAGE

abs(*i*)
int *i*;

DESCRIPTION

Abs returns the absolute value of its integer operand.

NOTES

Applying the **abs** function to the most negative integer generates a result that is the most negative integer. That is,

abs(0x80000000)

returns 0x80000000 as a result.

RELATED INFORMATION

floor(3M)

NAME

atof, atoi, atol – convert ASCII to numbers

USAGE

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

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

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

DESCRIPTION

These functions convert the string that *nptr* points to into floating, integer, and long integer representation, respectively. The first character that the function does not recognize ends the string.

Atof recognizes an optional string of spaces, then an optional sign, then a string of digits which may contain a decimal point, then an optional "e" or "E", followed by an optionally signed integer.

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

NOTES

None of these functions has provisions for overflow.

RELATED INFORMATION

scanf(3S)

NAME

bcopy, bcmp, bzero, ffs – bit and byte string operations

USAGE

bcopy(*b1*, *b2*, *length*)
char **b1*, **b2*;
int *length*;

bcmp(*b1*, *b2*, *length*)
char **b1*, **b2*;
int *length*;

bzero(*b*, *length*)
char **b*;
int *length*;

ffs(*i*)
int *i*;

DESCRIPTION

The functions **bcopy**, **bcmp**, and **bzero** operate on variable length strings of bytes. They do not check for null bytes as the routines in **string**(3) do.

Bcopy copies *length* bytes from string *b1* to string *b2*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* zero bytes in the string *b1*.

Ffs returns the index of the first bit set in its argument. A zero return indicates a zero argument. Bits are numbered starting at 1.

NOTES

The **bcmp** and **bcopy** routines take parameters in reverse order from **strcmp** and **strcpy**. For example,

strcpy (foo, bar)

copies *foo* to *bar*, while

bcpy (foo, bar, 3)

copies *bar* to *foo*.

NAME

crypt, encrypt – a one-way hashing encryption algorithm

USAGE

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

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

DESCRIPTION

The password encryption function, **crypt**, is based on a one-way hashing encryption algorithm with variations partly intended to frustrate hardware implementations of a key search.

The *key* parameter represents a user's typed password. The *salt* parameter is a two-character string chosen from the set [a-zA-Z0-9./]; this 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.

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 0 and 1. The argument array is modified in place, becoming a similar array that represents the bits of the argument after exposure to the hashing algorithm using the key set by **crypt**.

Note: Per international agreement not to export encryption devices, the standard UNIX system decryption methods are not supported on the DOMAIN/IX system.

NOTES

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

RELATED INFORMATION

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

NAME

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

USAGE

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

#include <sys/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 denoted by *clock*, such as the value returned by **time**(2), into ASCII and returns a pointer to a 26-character string in the following form.

Thu May 29 10:32:03 1986\n\0

All fields have constant width. **Localtime** and **gmtime** return pointers to structures containing the individual components of the time. **Localtime** corrects for the time zone and daylight savings time (if necessary); **gmtime** converts directly to GMT, which is the time DOMAIN/IX uses. **Asctime** converts a time from the structures to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file 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 minus (-) 1900, day of year (0-365), and a flag that is non-zero if daylight savings time is in effect.

When local time is necessary, the program consults the system to determine the time zone and whether the U.S.A., Australian, Eastern European, Middle European, or Western European daylight savings time adjustment is appropriate. The program understands some of the peculiarities in time conversion over the past 10-20 years; if necessary, this understanding 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 zero, the standard zone name is used; otherwise, the Daylight Savings Zone. 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 Afghanistan is four and a half hours ahead of GMT. This call would produce the string **GMT+4:30.**

NOTES

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

RELATED INFORMATION

gettimeofday(2), time(3C)

NAME

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

USAGE

#include <ctype.h>

isalpha(*c* **)**

.
.
.

isascii(*c* **)**

DESCRIPTION

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

isalpha	c is a letter
isupper	c is an uppercase letter
islower	c is a lowercase 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).

NAME

opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

USAGE

#include <sys/dir.h>

DIR *opendir(*filename***)**
char **filename*;

struct direct *readdir(*dirp***)**
DIR **dirp*;

long telldir(*dirp***)**
DIR **dirp*;

seekdir(*dirp*, *loc***)**
DIR **dirp*;
long *loc*;

rewinddir(*dirp***)**
DIR **dirp*;

closedir(*dirp***)**
DIR **dirp*;

DESCRIPTION

Opendir opens the directory named by *filename* and associates a "directory stream" with it. **Opendir** returns a pointer that identifies the directory stream in subsequent operations. **Opendir** returns a NULL pointer if *filename* cannot be accessed, or if **malloc**(3) cannot allocate enough memory to hold the entire **DIR** structure.

Readdir returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory, or upon detecting an invalid **seekdir** operation.

Telldir returns the current location associated with the directory stream.

Seekdir sets the position of the next **readdir** operation on the directory stream. The new position reverts to the one associated with the directory stream when the **telldir** operation was performed. Values returned by **telldir** are good only for the lifetime of the **DIR** pointer from which they are derived. If the directory is closed and then reopened, the **telldir** value may be invalidated due to undetected directory compaction. It is safe to use a previous **telldir** value immediately after a call to **opendir** and before any calls to **readdir**.

Rewinddir resets the position of the named directory stream to the beginning of the directory.

Closedir closes the named directory stream and frees the structure associated with the **DIR** pointer.

EXAMPLE

Sample code that searches a directory for entry "name" is:

```
len = strlen(name);
dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(dirp);
                return FOUND;
        }
closedir(dirp);
return NOT_FOUND;
```

RELATED INFORMATION

open(2), **close**(2), **read**(2), **lseek**(2)

NAME

ecvt, fcvt, gcvt – output conversion

USAGE

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 to the string. 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 that *sign* points to is non-zero; otherwise, it is zero. The low-order digit is rounded.

Fcvt is similar to ecvt, except that the correct digit has been rounded for FORTRAN F-format output of the number of digits specified by *ndigits*.

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

NOTES

The return values point to static data that each call overwrites.

RELATED INFORMATION

printf(3)

NAME

end, etext, edata – last location in program

USAGE

extern end;
extern etext;
extern edata;

DESCRIPTION

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

When execution begins, the program break coincides with **end,** but it is reset by the routines **brk**(2), **malloc**(3), standard input/output **stdio**(3), the profile (**–p**) option of *cc*(1), and so on. The current value of the program break is reliably returned by calling **sbrk(0)**.

RELATED INFORMATION

brk(2)
malloc(3)

NAME

execl, execv, execle, execlp, execvp, exect, environ – execute a file

USAGE

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*[ ];

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

extern char ***environ*;

DESCRIPTION

These routines provide various interfaces to the execve system call. Refer to execve(2) for a full description of their properties; only brief descriptions are provided here.

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.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[0], *arg*[1], ..., address null-terminated strings. In most cases, *arg*[0] is the name of the file.

Two interfaces are available. **Execl** is useful when a known *name* with known arguments is being called; the arguments to **execl** are the character strings that comprise the file (*name*) and the arguments. The first argument is usually the same as the filename (or its last component). A zero argument ends the argument list.

The **execv** version is useful when the number of arguments is not known in advance; the arguments to **execv** include 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 zero pointer.

The exect version is used when the executed file is to be manipulated with **ptrace(2)**. It forces the child to stop after executing its first instruction. The parent (which must expect to trace the child) may then adjust the child's state.

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. 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 zero.

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an equals sign (=), and a null-terminated value. The array of pointers is terminated by a null pointer. The shell passes an environment entry for each global shell variable that is defined when the program is called. The C run-time start-off routine places a copy of *envp* in the global cell **environ**, which **execv** and **execl** use to pass the environment to any subprograms executed by the current program.

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**

DIAGNOSTICS

A return constitutes the diagnostic if any of the following hold true:

- *name* cannot be found
- *name* is not executable
- *name* is not an object module
- maximum memory was exceeded
- the arguments require too much space

The return value is -1. Even if the caller is the super-user, at least one of the execute-permission bits must be set for a file to be executed.

RELATED INFORMATION

execve(2), fork(2), csh(1)

NAME

exit – terminate a process after flushing any pending output

USAGE

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

DESCRIPTION

Exit terminates a process after calling the standard I/O library function **_cleanup** to flush any buffered output. Exit never returns.

RELATED INFORMATION

exit(2)

NAME

frexp, ldexp, modf – split into mantissa and exponent

USAGE

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 (indirectly through *eptr*) an integer *n* such that *value* = *x*2**n.*

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

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

NAME

getenv – get the value of an environment variable

USAGE

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

DESCRIPTION

Getenv searches through the list of environment variables for a string of the form:

name=value

If it finds an entry, **getenv** returns a pointer to the null-terminated string *value*. If it cannot find an entry for *name*, **getenv** returns the value zero (NULL).

RELATED INFORMATION

execve(2)

NAME

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

USAGE

#include <grp.h>

struct group *getgrent( )

struct group *getgrgid(*gid* **)**
int *gid***;**

struct group *getgrnam(*name* **)**
char **name***;**

setgrent( )

endgrent( )

DESCRIPTION

Getgrent, getgrgid and **getgrnam** return pointers to an object with the following structure, which contains 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;
};

struct group *getgrent(), *getgrgid(), *getgrnam();
```

The members of this structure are:

gr_name The name of the group.

gr_passwd The encrypted password of the group (always null on DOMAIN/IX Systems).

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.

NOTES

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

On DOMAIN/IX Systems, */etc/group* is built from registry information by the program **crpasswd(8)**.

DIAGNOSTICS

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

FILES

/etc/group the group file

RELATED INFORMATION

getlogin(3), getpwent(3), group(5), crpasswd(8)

NAME

getlogin – get log-in name

USAGE

char *getlogin()

DESCRIPTION

Getlogin returns a pointer to the user's log-in name. It may be used in conjunction with **getpwnam** to locate the correct password file entry when several log-in names share the same user ID.

If **getlogin** is called within a process that is not attached to a terminal, it returns NULL. To determine the log-in name, first call **getlogin**; if it fails, call **getpwuid(getuid())**.

NOTES

The return values point to static data, which each call overwrites.

DIAGNOSTICS

Returns NULL (zero) if name is not found.

RELATED INFORMATION

getpwent(3), **getgrent**(3), **getpwuid**(3)

NAME
getpass – read a password

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

DESCRIPTION
Getpass prompts for a password with the null-terminated string *prompt*, then disables echoing of input characters. On DOMAIN Systems, **getpass** reads a password from an input pad (the local equivalent of */dev/tty*) or, if the standard input is an SIO line, from */dev/sio?*. If neither of these files can be read, **getpass** reads a password from the standard input.

Getpass returns a pointer to a null-terminated string of at most eight characters.

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

NAME

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

USAGE

#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. It contains the broken-out fields of a line in the password file.

```
struct  passwd { /* see getpwent(3) */
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        int     pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};

struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

The fields **pw_quota** and **pw_comment** are unused. The rest are described in the manual entry for **passwd(5)**.

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

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

NOTES

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

On DOMAIN/IX Systems, */etc/passwd* is built from registry information by the program **crpasswd(8)**.

DIAGNOSTICS

Null pointer (zero) returned on EOF or error.

FILES

/etc/passwd the password file

RELATED INFORMATION

getlogin(3), getgrent(3), passwd(5), crpasswd(8)

NAME

getwd – get current working directory pathname

USAGE

char *getwd(*pathname*)
**char ***pathname*;

DESCRIPTION

Getwd copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

NOTES

Maximum pathname length is MAXPATHLEN characters (1024).

DIAGNOSTICS

Getwd returns zero and places a message in *pathname* if an error occurs.

NAME

insque, remque – insert or remove an element in a queue

USAGE

```
struct qelem {
        struct  qelem *q_forw;
        struct  qelem *q_back;
        char    q_data[ ];
};

insque( elem, pred)
struct qelem *elem, *pred;

remque( elem)
struct qelem *elem;
```

DESCRIPTION

Insque and **remque** manipulate queues built from doubly linked lists. Each element in the queue must be in the form of **struct qelem**. **Insque** inserts *elem* in a queue immediately after *pred*; **remque** removes an entry *elem* from a queue.

NAME

malloc, free, realloc, calloc, alloca – memory allocator

USAGE

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

free(*ptr***)**
char **ptr*;

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

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

char *alloca(*size***)**
int *size*;

DESCRIPTION

Malloc and **free** provide simple, general-purpose memory allocation functions. **Malloc** returns a pointer to a block of at least *size* bytes that begins 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.

Malloc maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls **sbrk** (see **brk**(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block to which *ptr* points, 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.

In order to be compatible with older versions, **realloc** also works if *ptr* points to a block freed since the last call of **malloc**, **realloc**, or **calloc**; sequences of **free**, **malloc**, and **realloc** have been used in the past to attempt storage compaction. This procedure is no longer recommended.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initially filled with zeros.

Alloca allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Each of the allocation routines returns a pointer to space suitably aligned for storage of any type of object.

NOTES

In previous versions of DOMAIN/IX, **malloc** incorrectly added space for a terminal null when allocating storage for a string. This behavior has changed at this release. **Malloc** no longer allocates the extra byte of storage, so programs that failed to allow for the null at the end of a string are likely to fail with a reference to an illegal address.

If the space assigned by **malloc** is overrun, or if a random number is handed to **free**, problems will result.

When **realloc** returns zero, the block that *ptr* points to may be destroyed.

Alloca is machine-dependent; its use is discouraged.

DIAGNOSTICS

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

RELATED INFORMATION

brk(2), sbrk(2), environ(7)

NAME

mktemp – make a unique filename

USAGE

char *mktemp(*template*)
char **template*;

DESCRIPTION

Mktemp generates and returns the address of a unique, usually temporary, filename based on *template*. The *template* should look like a filename with six trailing Xs, for example

```
t = mktemp("/tmp/tfXXXXXX");
```

The Xs will be replaced with the current process ID and a unique letter.

NOTES

It is possible to run out of letters.

RELATED INFORMATION

getpid(2)

NAME

perror, sys_errlist, sys_nerr – system error messages

USAGE

perror(*s* **)**
char **s*;

int sys_nerr;
char *sys_errlist[ ];

DESCRIPTION

Perror produces a short error message on the standard error file that describes the error that a C program encountered during its most recent call to the system. The argument string *s* is printed first, followed by a colon, the message, and a new-line. The argument string is the name of the program that caused the error. The error number is taken from the external variable *errno*, which is set when errors occur.

The vector of message strings, **sys_errlist**, is provided to simplify the message formats. Use *errno* as an index into this table to get the message string without the new-line. **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.

NOTES

Errno is only set when an error occurs. It is not cleared when a valid call is made.

RELATED INFORMATION

psignal(3)

NAME

popen, pclose – initiate I/O to and from a process

USAGE

#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 that contain a shell command line and an I/O mode, respectively. The I/O mode is either "r" for reading or "w" for writing. **Popen** creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by **popen** should be closed by **pclose**, which waits for the process associated with it to terminate and returns the exit status of the command.

Because open files are shared, an "r" command may act as an input filter, and a "w" as an output filter.

NOTES

Buffered reading before opening an input filter may leave the standard input of that filter in the wrong position. Similar problems with an output filter may be forestalled by careful buffer flushing with **fflush**; see fclose(3).

Popen always calls **sh**, never **csh**.

DIAGNOSTICS

Popen returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns -1 if *stream* is not associated with a command opened by **popen**.

RELATED INFORMATION

pipe(2), fopen(3S), fclose(3S), system(3), wait(2), sh(1)

NAME

psignal, sys_siglist – system signal messages

USAGE

psignal(*sig*, *s*)
unsigned *sig*;
char *s;

char *sys_siglist[ ];

DESCRIPTION

Psignal produces a short message on the standard error file describing the indicated *sig*nal. The message consists of the argument string *s*, a colon, the name of the signal, and a newline. In practice, *s* is usually the name of the program that incurred the signal. The signal number should be one of those found in */usr/include/signal.h.*

A vector of message strings, **sys_siglist**, is provided to simplify variant formatting of signal names. The signal number can be used as an index into this table to get the signal name without the newline. The "define NSIG" defined in *signal.h* is the number of messages provided for in the table; it should be checked, because assignment of signals to numbers may change, and new signals may be added to the system before they are added to the table.

RELATED INFORMATION

sigvec(2), perror(3)

NAME

qsort – quicker sort

USAGE

qsort(*base*, *nel*, *width*, *compar***)**
char **base***;**
int (**compar***)( );**

DESCRIPTION

Qsort is an implementation of a quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; and the third is the width of an element in bytes.

The last argument is the name of the comparison routine to be called; the routine is called with two arguments that are pointers to the two elements being compared. The routine must return an integer less than, equal to, or greater than zero, depending on whether the first argument (i.e., the first element being compared) is to be considered less than, equal to, or greater than the second.

RELATED INFORMATION

sort(1)

NAME

random, srandom, initstate, setstate – better random number generator and associated routines

USAGE

```
long random( )

srandom( seed)
int seed;

char *initstate( seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate( state)
char *state;
```

DESCRIPTION

Random implements a non-linear additive feedback random number generator. It uses a default table of 31 long integers to return successive pseudo-random numbers in the range from 0 to 2^{31} - 1. The period of this random number generator is very large, approximately $16*(2^{31}-1)$.

Random/srandom have (almost) the same calling sequence and initialization properties as **rand/srand.** The difference is that **rand**(3) produces a much less random sequence — in fact, the low dozen bits generated by **rand** go through a cyclic pattern. All the bits generated by **random** are usable. For example,

```
random( )&01
```

will produce a random binary value.

Unlike **srand**, **srandom** does not return the old seed, because the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators). Like **rand**(3), however, **random** will produce a sequence of numbers that can be duplicated by calling **srandom** with **1** as the seed.

The **initstate** routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by **initstate** to decide how sophisticated a random number generator it should use — the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The *seed* for the initialization (which specifies a starting point for the random number sequence and provides for restarting at the same point) is also an argument. **Initstate** returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate** routine provides for rapid switching between states. **Setstate** returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to **initstate** or **setstate.**

Once a state array has been initialized, it may be restarted at a different point, either by calling **initstate** (with the desired seed, the state array, and its size) or by calling both **setstate** (with the state array) and **srandom** (with the desired seed). The advantage of calling both **setstate** and **srandom** is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

NOTES

Random is about two thirds as fast as **rand**(3C). However, **random** does produce a more random number or numbers.

DIAGNOSTICS

If **initstate** is called with less than 8 bytes of state information, or if **setstate** detects that the state information has been garbled, error messages are printed on the standard error output.

RELATED INFORMATION

rand(3C)

NAME

re_comp, re_exec – regular expression handler

USAGE

char *re_comp(*s*) char *s;

re_exec(*s*) char *s;

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. **Re_exec** checks the argument string against the last string passed to **re_comp.**

Re_comp returns zero if the string *s* was compiled successfully; otherwise it returns a string containing an error message. If **re_comp** is passed zero or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, zero if the string *s* failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

A string passed to either **re_comp** or **re_exec** may have trailing or embedded newline characters, and is null-terminated. With that exception, recognized regular expressions are the ones described in the manual entry for **ed**(1).

DIAGNOSTICS

Re_exec returns -1 for an internal error.

Re_comp returns one of the following strings if an error occurs:

No previous regular expression,
Regular expression too long
unmatched \(
missing]
too many \(\) *pairs*
unmatched \)

RELATED INFORMATION

ed(1), ex(1), grep(1), sed(1)

NAME

scandir – scan a directory

USAGE

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

scandir(*dirname*, *namelist*, *select*, *compar*)
char **dirname*;
struct direct *(**namelist*[ ]);
int (**select*)( );
int (**compar*)( );

alphasort(*d1*, *d2*)
struct direct ***d1*, ***d2*;

DESCRIPTION

Scandir reads the directory *dirname* and builds (using **malloc**(3)) an array of pointers to directory entries. It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user-supplied subroutine that **scandir** calls to select the entries to be will be included in the array. The *select* routine is passed a pointer to a directory entry, and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user-supplied subroutine that is passed to **qsort**(3) to sort the completed array. If this pointer is null, the array is not sorted. **Alphasort** is a routine which can be used for the *compar* parameter. It sorts the array alphabetically.

The memory allocated for the array can be deallocated with **free** (see **malloc**(3)) by freeing each pointer in the array and then the array itself.

DIAGNOSTICS

Returns -1 if the directory cannot be opened for reading or if **malloc**(3) cannot allocate enough memory to hold all the data structures.

RELATED INFORMATION

directory(3), **malloc**(3), **qsort**(3),

NAME

setjmp, longjmp – non-local goto

USAGE

#include <setjmp.h>

setjmp(*env***)**
jmp_buf *env***;**

longjmp(*env***,** *val***)**
jmp_buf *env***;**

_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 a value of zero.

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**. **Setjmp** itself must not have returned in the interim. All accessible data has values as of the time **longjmp** was called.

Setjmp and **longjmp** save and restore the signal mask **sigsetmask(2)**, while **_setjmp** and **_longjmp** manipulate only the stack and registers.

RELATED INFORMATION

sigvec(2), sigstack(2), signal(3C)

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

USAGE

setuid(*uid***)**
seteuid(*euid***)**
setruid(*ruid***)**

setgid(*gid***)**
setegid(*egid***)**
setrgid(*rgid***)**

DESCRIPTION

Setuid (**setgid**) sets both the real and effective user ID (group ID) of the current process to the ID specified in the function.

Seteuid (**setegid**) sets the effective user ID (group ID) of the current process.

Setruid (**setruid**) sets the real user ID (group ID) of the current process.

Only the super-user may use these calls, unless the argument is the real or effective ID of the caller.

DIAGNOSTICS

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

RELATED INFORMATION

setreuid(2), setregid(2), getuid(2), getgid(2)

NAME

sleep – suspend execution for interval

USAGE

sleep(*seconds* **)**
unsigned *seconds*;

DESCRIPTION

Sleep suspends the current process from execution for the prescribed number of *seconds*. The actual suspension time may be up to 1 second less than that requested, since scheduled wakeups occur at fixed 1-second intervals, which may be further extended by an arbitrary amount because of other system activity.

The routine is implemented by setting an interval timer and pausing until it times out. The previous state of this timer is saved and restored. If the **sleep** interval requested exceeds the time remaining on the previous timer, the process sleeps only until that timer times out (the signal is sent 1 second later).

RELATED INFORMATION

setitimer(2), sigpause(2)

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex – string operations

USAGE

```
#include <strings.h>

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 zero, according to whether *s1* is lexicographically greater than, equal to, or less than *s2*. **Strncmp** makes the same comparison but looks at a maximum of *n* characters.

Strcpy copies string *s2* to *s1*, stopping after the null character has been moved. **Strncpy** copies exactly *n* characters, truncating or null-padding *s2;* the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

Strlen returns the number of non-null characters in *s*.

Index (rindex) returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

NAME

swab – swap bytes

USAGE

swab(*from*, *to*, *nbytes*)
char **from*, **to*;

DESCRIPTION

Swab copies *nbytes* bytes from a place pointed to by *from* to the position specified by *to*, exchanging adjacent even and odd bytes. It is useful when moving binary data among various machines.

NOTES

Nbytes should be even.

NAME

system – issue a shell command

USAGE

system(*string*)
char **string*;

DESCRIPTION

System causes *string* to be sent to sh(1) as input, as if *string* had been typed at a shell prompt by a user. The current process waits until the shell has completed, then returns the exit status of the shell.

DIAGNOSTICS

Exit status 127 indicates that the shell couldn't be executed.

RELATED INFORMATION

sh(1), exec(2)

NAME

ttyname, isatty – find name of a terminal

USAGE

char *ttyname(*filedes***)**

isatty(*filedes***)**

DESCRIPTION

Ttyname returns a pointer to the null-terminated pathname of the terminal device associated with file descriptor *filedes* (this is a system file descriptor and has nothing to do with the standard I/O FILE typedef).

Isatty returns 1 if *filedes* is associated with a terminal device; otherwise, it returns zero.

NOTES

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

FILES

*/dev/** various devices

DIAGNOSTICS

Ttyname returns a null pointer (zero) if *filedes* does not describe a terminal device in directory */dev*.

RELATED INFORMATION

ioctl(2)

NAME
valloc – aligned memory allocator

USAGE
char *valloc(*size* **)**
unsigned *size***;**

DESCRIPTION
Valloc allocates *size* bytes, aligned on a page boundary. It is implemented by calling **malloc(3)** with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

DIAGNOSTICS
Valloc returns a null pointer (zero) if there is no available memory, or if the arena has been detectably corrupted by storing outside the bounds of a block.

NAME

varargs – variable argument list

USAGE

#include <varargs.h>

function(**va_alist**)
va_dcl
va_list *pvar*;
va_start(*pvar***)**;
f = va_arg(*var*, *type*);
va_end(*pvar***)**;

DESCRIPTION

This set of macros provides a way to write portable procedures that accept variable argument lists. Routines with variable argument lists (such as **printf**(3)) that do not use **varargs** are inherently difficult to port, since different machines use different argument-passing conventions.

Va_alist is used in a function header to declare a variable argument list.

Va_dcl is a declaration for **va_alist**. Note that there is no semicolon after **va_dcl.**

Va_list is a type that can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

Va_start(*pvar***)** is called to initialize *pvar* to the beginning of the list.

Va_arg(*pvar*, *type***)** will return the next argument in the list pointed to by *pvar*. *Type* is the expected type of the argument. Different types can be mixed, but the routine should know what type of argument is expected, since it cannot be determined at runtime.

Va_end(*pvar***)** is used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end,** are possible.

NOTES

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, **execl** passes a zero to signal the end of the list. **Printf** can tell from the format how many arguments are supposed to be there.

EXAMPLE

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[100];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while (args[argno++] = va_arg(ap, char *))
                ;
        va_end(ap);
        return execv(file, args);
}
```

NAME
intro – introduction to compatibility library functions

DESCRIPTION
These functions constitute a compatibility library. They are part of */lib/clib*, and are automatically loaded as needed by the C compiler cc(1). Many of these routines have been rendered obsolete by newer ones. They are included here so that older programs will compile and run, but their use in new programs should, for the most part, be avoided. Manual entries for "obsolete" functions also name the newer, preferred, function.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
alarm	alarm.3c	schedule signal after specified time
ftime	time.3c	get date and time
getpw	getpw.3c	get name from uid
gtty	stty.3c	set and get terminal state (defunct)
nice	nice.3c	set program priority
pause	pause.3c	stop until signal
rand	rand.3c	random number generator
signal	signal.3c	simplified software signal facilities
srand	rand.3c	random number generator
stty	stty.3c	set and get terminal state (defunct)
time	time.3c	get date and time
times	times.3c	get process times
utime	utime.3c	set file times

NAME

alarm – schedule signal after specified time (obsolete)

USAGE

alarm(*seconds* **)**
unsigned *seconds*;

DESCRIPTION

This interface has been made obsolete by **setitimer**(2).

Alarm causes the signal SIGALRM (see **signal**(3C)), to be sent to the invoking process after the number of *seconds* specified by the argument. Unless caught or ignored by the program, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If *seconds* is set to zero, any pending alarm request is cancelled. Because of scheduling delays, when the signal is caught, the program may not resume execution immediately. The largest legal value for *seconds* is 2147483647.

RETURN VALUE

The return value is the amount of time remaining until any alarm that may have been pending.

RELATED INFORMATION

sigpause(2), **sigvec**(2), **signal**(3C), **sleep**(3)

NAME

getpw – get name from user ID (obsolete)

USAGE

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

DESCRIPTION

Getpw has been made obsolete by **getpwuid**(3).

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

FILES

/etc/passwd the password file

DIAGNOSTICS

Returns non-zero on an error.

RELATED INFORMATION

getpwent(3)

NAME

nice – set program priority (obsolete)

USAGE

nice(*incr***)**

DESCRIPTION

This interface has been made obsolete by **setpriority**(2).

The amount *incr* increases the scheduling priority of the process. Positive priorities get less service than normal. Priority 10 allows long-running programs to operate without adversely affecting the entire system's performance.

The priority is limited to the range -20 (most urgent) to 20 (least).

The priority of a process passes to a child process spawned by **fork** (2). To recall a privileged process to normal priority from an unknown state, call **nice** with arguments -40 (goes to priority -20 because of truncation), 20 (to get to zero), then zero successively.

RELATED INFORMATION

nice(1), **setpriority**(2), **fork**(2), **renice**(8)

NAME

pause – stop until signal

USAGE

pause( )

DESCRIPTION

Pause never returns normally. It causes a program to give up control and wait for a signal from **kill(2)** or an interval timer; see **setitimer(2)**. When a signal handler that was started during a **pause** terminates, the **pause** call will return.

RETURN VALUE

This function always returns -1.

ERRORS

Pause always sets *errno* to:

[EINTR]   The call was interrupted.

RELATED INFORMATION

kill(2), select(2), sigpause(2)

NAME

rand, srand – random number generator (obsolete)

USAGE

srand(*seed*)
int *seed*;

rand( )

DESCRIPTION

The newer **random**(3) should be used in new applications; **rand** remains for compatibility.

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

The generator is reinitialized by calling **srand** with 1 as argument. It can be set to a random starting point by calling **srand** with any integer as an argument.

RELATED INFORMATION

random(3)

NAME

signal – simplified software signal facilities

USAGE

#include <signal.h>

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

DESCRIPTION

Signal is a simplified interface to the more general **sigvec(2)** facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see **tty**(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. The SIGKILL and SIGSTOP signals cannot be caught or ignored. **Signal** allows all other signals to be ignored, or to generate an interrupt to a specified location. The following is a list of all signals with names as in the include file **<signal.h>**:

SIGHUP	1	hang-up
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction
SIGTRAP	5	trace trap
SIGIOT	6	IOT instruction
SIGEMT	7	EMT instruction
SIGFPE	8	floating-point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	death of a child
SIGAPOLLO	19	DOMAIN System fault with no UNIX equivalent
SIGSTOP	20†	stop, cannot be caught, held, or ignored

SIGTSTP	21†	stop signal generated from keyboard
SIGCONT	22•	continue after stop
SIGCHLD	23•	child status has changed
SIGTTIN	24†	background read attempted from control terminal
SIGTTOU	25†	background write attempted to control terminal
SIGIO	26	I/O is possible on a descriptor
SIGTINT	26	input record is available at control terminal
SIGXCPU	27	cpu time limit exceeded
SIGXFSZ	28	file size limit exceeded
SIGVTALRM	29	virtual time alarm
SIGPROF	30	profiling timer alarm
SIGURG	31•	urgent condition present on socket

If *func* is SIG_DFL, the default action for signal **sig** is reinstated. This default is termination, except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL; signals marked with † cause the process to stop. If *func* is SIG_IGN, the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.

During certain system calls, if a caught signal occurs and the call terminates prematurely, the call is automatically restarted. In particular, this can occur during a **read** or **write**(2) on a slow device (such as a terminal) and during a **wait**(2).

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

After a **fork**(2) or **vfork**(2) the child inherits all signals. Execve(2) resets all signals caught to the default action; ignored signals are not affected.

NOTES

DOMAIN systems send the signal SIGAPOLLO whenever a fault occurs that is not otherwise mapped into a signal. Typical generators of SIGAPOLLO include network failures, display-acquire timeouts, and disk full errors.

The handler routine can be declared:

handler(sig, code, scp)

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. *Code* is a 32-bit value; one of the values listed above or, if the signal is SIGAPOLLO, the DOMAIN System status code describing the fault. To generate a list of DOMAIN System status codes and brief explanations of their meanings, run the command **/systest/ssr_util/all_stcode**. *Scp* is a pointer to the **struct sigcontext** used by the system to restore the process context from before the signal. Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in <*signal.h*>:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	
Protection violation	SIGBUS	
Reserved instruction	SIGILL	ILL_RESAD_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_PRIVIN_FAULT
Reserved addressing	SIGILL	ILL_RESOP_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	

RETURN VALUE

The previous action is returned on a successful call. Otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

Signal will fail and no action will take place if one of the following occur:

[EINVAL] *Sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

RELATED INFORMATION

kill(1), kill(2), sigvec(2), sigblock(2), sigsetmask(2), sigpause(2) sigstack(2), setjmp(3), tty(4)

NAME

stty, gtty – set/get terminal state (obsolete)

USAGE

#include <sgtty.h>

stty(*fd*, *buf*)
int *fd*;
**struct sgttyb ****buf*;

gtty(*fd*, *buf*)
int *fd*;
**struct sgttyb ****buf*;

DESCRIPTION

This interface has been made obsolete by **ioctl**(2).

Stty sets the state of the terminal associated with *fd*. **Gtty** retrieves the state of the terminal associated with *fd*. To set the state of a terminal, the call must have write permission.

The **stty** call is actually

ioctl(*fd*, TIOCSETP, *buf*)

and the **gtty** call is

ioctl(*fd*, TIOCGETP, *buf*)

See **ioctl**(2) and **tty**(4) for explanations.

RETURN VALUE

A successful call returns zero. A failed call returns -1 and sets *errno*.

RELATED INFORMATION

ioctl(2)

NAME

time, ftime – get date and time (obsolete)

USAGE

long time(0)

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

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

DESCRIPTION

These interfaces have been made obsolete by **gettimeofday(2)**.

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>:

```
struct timeb
{
        time_t    time;
        unsigned short millitm;
        short     timezone;
        short     dstflag;
};
```

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

RELATED INFORMATION

date(1), gettimeofday(2), settimeofday(2), ctime(3C)

NAME

times – get process times

USAGE

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

times(*buffer***)**
struct tms ***buffer*;**

DESCRIPTION

Times returns time-accounting information for the current process and for any terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is 60.

This is the structure returned by **times**:

```
struct tms {
        time_t tms_utime;          /* user time */
        time_t tms_stime;          /* system time */
        time_t tms_cutime;         /* user time, children */
        time_t tms_cstime;         /* system time, children */
};
```

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

On DOMAIN Systems, the system time is always returned as 0, since it is considered part of the user time.

RELATED INFORMATION

time(1), wait3(2), time(3c)

NAME

utime – set file times (obsolete)

USAGE

#include <sys/types.h>

utime(*file*, *timep*)
char *_file_;
time_t *timep*[2];

DESCRIPTION

This interface has been made obsolete by **utimes(2)**.

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

The caller must be the owner of the file or the super-user. The "inode-changed" time of the file is set to the current time.

RELATED INFORMATION

utimes(2), stat(2)

NAME

intro – introduction to mathematical library functions

DESCRIPTION

These math functions are a part of */lib/clib*. Declarations for these functions may be obtained from the include file <*math.h*>.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
acos	sin.3m	trigonometric functions
asin	sin.3m	trigonometric functions
atan	sin.3m	trigonometric functions
atan2	sin.3m	trigonometric functions
cabs	hypot.3m	Euclidean distance
ceil	floor.3m	absolute value, floor, ceiling functions
cos	sin.3m	trigonometric functions
cosh	sinh.3m	hyperbolic functions
exp	exp.3m	exponential, logarithm, power, square root
fabs	floor.3m	absolute value, floor, ceiling functions
floor	floor.3m	absolute value, floor, ceiling functions
gamma	gamma.3m	log gamma function
hypot	hypot.3m	Euclidean distance
j0	j0.3m	bessel functions
j1	j0.3m	bessel functions
jn	j0.3m	bessel functions
log	exp.3m	exponential, logarithm, power, square root
log10	exp.3m	exponential, logarithm, power, square root
pow	exp.3m	exponential, logarithm, power, square root
sin	sin.3m	trigonometric functions
sinh	sinh.3m	hyperbolic functions
sqrt	exp.3m	exponential, logarithm, power, square root
tan	sin.3m	trigonometric functions
tanh	sinh.3m	hyperbolic functions
y0	j0.3m	bessel functions
y1	j0.3m	bessel functions
yn	j0.3m	bessel functions

NAME

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

USAGE

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

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

double sqrt(x)
double x;
```

DESCRIPTION

Exp returns the exponential function of *x*.

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

Pow returns x^y.

Sqrt returns the square root of *x*.

DIAGNOSTICS

When the correct value would overflow, **exp** and **pow** return HUGE and sets *errno* to ERANGE.

Pow returns zero and sets *errno* to EDOM when the second argument is negative and not an integer, and when both arguments are zero.

Log returns zero when *x* is zero or negative; *errno* is set to EDOM.

Sqrt returns zero when *x* is negative; *errno* is set to EDOM.

RELATED INFORMATION

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

NAME

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

USAGE

#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.

RELATED INFORMATION

abs(3)

NAME

gamma – log gamma function

USAGE

```
#include <math.h>

double gamma(x)
double x;
```

DESCRIPTION

Gamma returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer **signgam.**

EXAMPLE

The following C program might be used to calculate Γ:

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

DIAGNOSTICS

HUGE is returned for negative integer arguments.

NOTES

There is no positive indication of error.

NAME

hypot, cabs – Euclidean distance

USAGE

```
#include &lt;math.h&gt;

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

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

DESCRIPTION

Hypot and **cabs** return

sqrt(x*x + y*y),

The functions include allowances for unwarranted overflows.

RELATED INFORMATION

exp(3M)

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

USAGE

```
#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 -HUGE and set *errno* to EDOM.

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

USAGE

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(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 arcsine in the range $-\pi/2$ to $\pi/2$.

Acos returns the arccosine in the range zero to π.

Atan returns the arctangent of x in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arctangent of x/y in the range $-\pi$ to π.

NOTES

The value of **tan** for arguments greater than about 2^{31} is meaningless.

DIAGNOSTICS

Arguments of magnitude greater than one cause **asin** and **acos** to return value zero; *errno* is set to EDOM. The value of **tan** at its singular points is HUGE, and *errno* is set to ERANGE.

NAME

sinh, cosh, tanh – hyperbolic functions

USAGE

#include <math.h>

double sinh(*x*)

double cosh(*x*)
double *x*;

double tanh(*x*)
double *x*;

DESCRIPTION

These functions compute the specified hyperbolic functions for a real *x*.

DIAGNOSTICS

Sinh and **cosh** return +/- HUGE when the correct value would overflow.

NAME

intro – introduction to network library functions

DESCRIPTION

This section describes functions that are applicable to the DARPA Internet network.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
endhostent	gethostent.3n	get network host entry
endnetent	getnetent.3n	get network entry
endprotoent	getprotoent.3n	get protocol entry
endservent	getservent.3n	get service entry
gethostbyaddr	gethostent.3n	get network host entry
gethostbyname	gethostent.3n	get network host entry
gethostent	gethostent.3n	get network host entry
getnetbyaddr	getnetent.3n	get network entry
getnetbyname	getnetent.3n	get network entry
getnetent	getnetent.3n	get network entry
getprotobyname	getprotoent.3n	get protocol entry
getprotobynumber	getprotoent.3n	get protocol entry
getprotoent	getprotoent.3n	get protocol entry
getservbyname	getservent.3n	get service entry
getservbyport	getservent.3n	get service entry
getservent	getservent.3n	get service entry
htonl	byteorder.3n	convert values between host and network byte order
htons	byteorder.3n	convert values between host and network byte order
inet_addr	inet.3n	Internet address manipulation routines
inet_lnaof	inet.3n	Internet address manipulation routines
inet_makeaddr	inet.3n	Internet address manipulation routines
inet_netof	inet.3n	Internet address manipulation routines
inet_network	inet.3n	Internet address manipulation routines
ntohl	byteorder.3n	convert values between host and network byte order
ntohs	byteorder.3n	convert values between host and network byte order
sethostent	gethostent.3n	get network host entry
setnetent	getnetent.3n	get network entry
setprotoent	getprotoent.3n	get protocol entry
setservent	getservent.3n	get service entry

NAME

htonl, htons, ntohl, ntohs – convert values between host and network byte order

USAGE

#include <sys/types.h>
#include <netinet/in.h>

netlong = **htonl(** *hostlong***);**
u_long *netlong*, *hostlong*;

netshort = **htons(** *hostshort***);**
u_short *netshort*, *hostshort*;

hostlong = **ntohl(** *netlong***);**
u_long *hostlong*, *netlong*;

hostshort = **ntohs(** *netshort***);**
u_short *hostshort*, *netshort*;

DESCRIPTION

These routines handle conversion of 16- and 32-bit quantities between network byte order and host byte order. On some machines (including DOMAIN Systems), these routines are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by **gethostent(3N)** and **getservent(3N)**.

RELATED INFORMATION

gethostent(3N), getservent(3N)

NAME

gethostent, **gethostbyaddr**, **gethostbyname**, **sethostent**, **endhostent** – get network host entry

USAGE

#include <netdb.h>

struct hostent *gethostent()

struct hostent *gethostbyname(*name***)**
char ***name*;**

struct hostent *gethostbyaddr(*addr*, *len*, *type***)**
char ***addr*;**
int *len*, *type***;**

sethostent(*stayopen***)**
int *stayopen*

endhostent()

DESCRIPTION

Gethostent, **gethostbyname**, and **gethostbyaddr** all return a pointer to an object with the following structure, which contains the separated fields of a line in the network host database, */etc/hosts*.

```
struct  hostent {
        char    *h_name;      /* official name of host */
        char    **h_aliases;  /* alias list */
        int     h_addrtype;   /* address type */
        int     h_length;     /* length of address */
        char    *h_addr;      /* address */
};
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.

h_addr A pointer to the network address for the host. Host addresses are returned in network byte order.

Gethostent reads the next line of the file, opening the file if necessary.

Sethostent opens and rewinds the file. If the *stayopen* flag is non-zero, the host database will not be closed after each call to **gethostent** (either directly, or indirectly through one of the other "gethost" calls).

Endhostent closes the file.

Gethostbyname and **gethostbyaddr** sequentially search from the beginning of the file until a matching host name or host address is found, or until EOF is encountered. Host addresses are supplied in network byte order.

NOTES

All information is kept in a static area, so it must be copied if you wish to save it. These functions only understand the Internet address format.

FILES

/etc/hosts list of known host systems

DIAGNOSTICS

Null pointer (zero) returned on EOF or error.

RELATED INFORMATION

hosts(5)

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

USAGE

```
#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname( name)
char *name;

struct netent *getnetbyaddr( net, addrtype)
long net;
int addrtype;

setnetent( stayopen)
int stayopen

endnetent()
```

DESCRIPTION

Getnetent, **getnetbyname**, and **getnetbyaddr** each return a pointer to an object with the following structure, which contains the various fields of a line in the network database, */etc/networks*.

```
struct  netent {
        char    *n_name;      /* official name of net */
        char    **n_aliases;  /* alias list */
        int     n_addrtype;   /* net number type*/
        long    n_net;        /* net number */
};
```

The members of this structure are:

n_name	The official name of the network.
n_aliases	A zero-terminated list of alternate names for the network.
n_addrtype	The type of the network number returned; currently only AF_INET.
n_net	The network number. Network numbers are returned in machine byte order.

Getnetent reads the next line of the file, opening the file if necessary.

Setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not be closed after each call to **getnetent** (either directly, or indirectly through one of the other "getnet" calls).

Endnetent closes the file.

Getnetbyname and **getnetbyaddr** search sequentially from the beginning of the file until a matching net name or net address is found or until EOF is encountered. Network numbers are supplied in host order.

NOTES

All information is kept in a static area, so it must be copied if you wish to save it. These functions only understand the Internet address format. If *addrtype* is supplied, it must be AF_INET.

DIAGNOSTICS

Null pointer (zero) returned on EOF or error.

FILES

/etc/networks database of reachable networks

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

USAGE

#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(*name* **)**
char **name*;

struct protoent *getprotobynumber(*proto* **)**
int *proto*;

setprotoent(*stayopen* **)**
int *stayopen*

endprotoent()

DESCRIPTION

Getprotoent, getprotobyname, and **getprotobynumber** each return a pointer to an object with the following structure, which contains the fields of a line in the network protocol database, */etc/protocols*.

```
struct  protoent {
        char    *p_name;      /* official name of protocol */
        char    **p_aliases;  /* alias list */
        long    p_proto;      /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero-terminated list of alternate names for the protocol.

p_proto The protocol number.

Getprotoent reads the next line of the file, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not close after each call to **getprotoent** (either directly or indirectly through one of the other "getproto" calls).

Endprotoent closes the file.

Getprotobyname and **getprotobynumber** search sequentially, from the beginning of the file, until a matching protocol name or number is found or until EOF is encountered.

NOTES

All information is kept in a static area, so you must copy it if you wish to save it. These functions only understand the Internet protocol (IP).

DIAGNOSTICS

Null pointer (zero) returned on EOF or error.

FILES

/etc/protocols	database of available protocols

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

USAGE

```
#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port;
char *proto;

setservent(stayopen)
int stayopen

endservent()
```

DESCRIPTION

Getservent, getservbyname, and getservbyport each return a pointer to an object with the following structure, which contains the fields of a line in the network services database, */etc/services*.

```
struct  servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        long    s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network-byte order.

s_proto The name of the protocol to use when contacting the service.

Getservent reads the next line of the file, opening the file if necessary.

Setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net database will not be closed after each call to **getservent** (either directly or indirectly through one of the other "getserv" calls).

Endservent closes the file.

Getservbyname and **getservbyport** search sequentially , from the beginning of the file, until a matching protocol name or port number is found or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

NOTES

All information is kept in a static area, so you must copy it if you wish to save it.

DIAGNOSTICS

Null pointer (zero) is returned on EOF or error.

FILES

/etc/services database of available services

RELATED INFORMATION

getprotoent(3N)

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

USAGE

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_addr(*cp* **)**
char **cp*;

int inet_network(*cp* **)**
char **cp*;

char *inet_ntoa(*in* **)**
struct inet_addr *in*;

struct in_addr inet_makeaddr(*net*, *lna* **)**
int *net*, *lna*;

int inet_lnaof(*in* **)**
struct in_addr *in*;

int inet_netof(*in* **)**
struct in_addr *in*;

DESCRIPTION

The routines **inet_addr** and **inet_network** interpret character strings that represent numbers expressed in the Internet standard "." (dot) notation, and return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine **inet_ntoa** takes an Internet address and returns an ASCII string that represents the address in "." notation. The routine **inet_makeaddr** takes an Internet network number and a local network address and constructs an Internet address from it. The routines **inet_netof** and **inet_lnaof** break apart Internet host addresses, and return the network number and local network address part, respectively.

All Internet addresses are returned in network byte order. All network numbers and local address parts are returned as machine-format integer values.

INTERNET ADDRESSES

Values specified using the "." notation take one of the following forms:

a.b.c.d When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

a.b.c When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".

a.b When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host".

a When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a "." notation may be decimal, octal, or hexadecimal, and are specified according to C language conventions. (I.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading zero implies octal. Numbers without a leading zero are interpreted as decimal).

NOTES

The string returned by **inet_ntoa** resides in a static memory area that is overwritten.

DIAGNOSTICS

Inet_addr and **inet_network** return the value -1 for erroneous requests.

RELATED INFORMATION

gethostent(3N), getnetent(3N)

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

USAGE

setuid(uid)
seteuid(euid)
setruid(ruid)

setgid(gid)
setegid(egid)
setrgid(rgid)

DESCRIPTION

Setuid (**setgid**) sets both the real and effective user ID (group ID) of the current process to the ID specified in the function.

Seteuid (**setegid**) sets the effective user ID (group ID) of the current process.

Setruid (*setruid*) sets the real user ID (group ID) of the current process.

Only the super-user may use these calls, unless the argument is the real or effective ID.

DIAGNOSTICS

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

RELATED INFORMATION

setreuid(2), setregid(2), getuid(2), getgid(2)

NAME

stdio – standard buffered input/output package

USAGE

#include <stdio.h>

FILE **stdin*;
FILE **stdout*;
FILE **stderr*;

DESCRIPTION

The functions described in section 3S constitute a user-level buffering scheme. The in-line macros getc and **putc**(3S) handle characters quickly. The higher level routines **gets, fgets, scanf, fscanf, fread, puts, fputs, printf, fprintf, fwrite** all use **getc** and **putc**; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to the defined type **FILE. Fopen**(3S) 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

The constant "pointer" NULL (0) designates no stream at all.

The 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 */usr/include/stdio.h*, which contains 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; they cannot be redeclared: **getchar, putc, putchar, feof, ferror, fileno.**

NOTES

The standard buffered functions do not interact well with certain other library and system functions, especially **vfork**(2) and **abort**(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 that a **FILE** pointer designates corrupt or otherwise unintelligible **FILE** data.

For purposes of efficiency, this implementation of the standard library has been changed to line buffer output to a terminal by default. It attempts to do this transparently by flushing the output whenever a **read**(2) from the standard input is necessary. This is almost always transparent, but may cause confusion or malfunctioning of programs which use standard I/O routines but use **read**(2) themselves to read from the standard input.

In cases where a large amount of computation is done after printing part of a line on an output terminal, it is necessary to **fflush**(3S) the standard output before going off and computing or else the output will not appear.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
clearerr	ferror.3s	stream status inquiries
fclose	fclose.3s	close or flush a stream
fdopen	fopen.3s	open a stream
feof	ferror.3s	stream status inquiries
ferror	ferror.3s	stream status inquiries
fflush	fclose.3s	close or flush a stream
fgetc	getc.3s	get character or word from stream
fgets	gets.3s	get a string from a stream
fileno	ferror.3s	stream status inquiries
fopen	fopen.3s	open a stream
fprintf	printf.3s	formatted output conversion
fputc	putc.3s	put character or word on a stream
fputs	puts.3s	put a string on a stream
fread	fread.3s	buffered binary input/output
freopen	fopen.3s	open a stream
fscanf	scanf.3s	formatted input conversion
fseek	fseek.3s	reposition a stream
ftell	fseek.3s	reposition a stream
fwrite	fread.3s	buffered binary input/output
getc	getc.3s	get character or word from stream
getchar	getc.3s	get character or word from stream
gets	gets.3s	get a string from a stream
getw	getc.3s	get character or word from stream
printf	printf.3s	formatted output conversion
putc	putc.3s	put character or word on a stream
putchar	putc.3s	put character or word on a stream

puts	puts.3s	put a string on a stream
putw	putc.3s	put character or word on a stream
rewind	fseek.3s	reposition a stream
scanf	scanf.3s	formatted input conversion
setbuf	setbuf.3s	assign buffering to a stream
sprintf	printf.3s	formatted output conversion
sscanf	scanf.3s	formatted input conversion
ungetc	ungetc.3s	push character back into input stream

RELATED INFORMATION

open(2), close(2), read(2), write(2), fread(3S), fseek(3S),

NAME

fclose, fflush – close or flush a stream

USAGE

```
#include <stdio.h>

int fclose(stream)
FILE *stream;

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

DESCRIPTION

Fclose forces 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 a call to exit(2).

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

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

RELATED INFORMATION

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

NAME
ferror, feof, clearerr, fileno – stream status inquiries

USAGE
#include <stdio.h>

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

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

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

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

DESCRIPTION
Feof returns a non-zero indicator when end of file (EOF) is read on the input *stream*; otherwise, it returns zero.

Ferror returns non-zero when an error has occurred in reading or writing on the named **stream**; if no error has occurred, it returns zero.

Clearerr resets the error indication on the named *stream*. Unless cleared by **clearerr**, the error indication lasts until the stream is closed.

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

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

RELATED INFORMATION
fopen(3S), open(2)

NAME

fopen, freopen, fdopen – open a stream

USAGE

#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 *filename* and associates a stream with it. **Fopen** returns a pointer that identifies the stream in later operations.

Type is a character string with one of the following values:

r open for reading

w create for writing

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

In addition, each *type* may be followed by a plus sign (+) to have the file opened for reading and writing. "r+" positions the stream at the beginning of the file, "w+" creates or truncates it, and "a+" positions it at the end. Both reads and writes may be used on read/write streams, with the limitation that an **fseek**, **rewind**, or reading an end-of-file must be used between a read and a write, or between a write and a read.

Freopen substitutes the file named for the open *stream*. It returns the original value of *stream*. The original stream is closed.

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

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

DIAGNOSTICS

Fopen and **freopen** return a null pointer if *filename* cannot be accessed.

RELATED INFORMATION

open(2), fclose(3)

NAME

fread, **fwrite** – buffered binary input/output

USAGE

#include <stdio.h>

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

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

DESCRIPTION

Fread reads, into an array referenced by *ptr*, *nitems* items of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

If *stream* is **stdin** and the standard output is line-buffered, then any partial output line will be flushed before any call is made to **read**(2) to satisfy the **fread**.

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

DIAGNOSTICS

Fread and **fwrite** return zero upon end of file (EOF) or error.

RELATED INFORMATION

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

NAME

fseek, ftell, rewind – reposition a stream

USAGE

#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 set at *offset* bytes from the beginning, the current position, or the end of the file, according to whether *ptrname* has been set to the value 0, 1, or 2, respectively.

Fseek cancels any of the effects of **ungetc**(3S).

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

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

DIAGNOSTICS

Fseek returns -1 on an unsuccessful seek.

RELATED INFORMATION

lseek(2), **fopen**(3S)

NAME

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

USAGE

#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 input *stream*.

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

The function **fgetc** operates like **getc**, and may be used to save object text.

Getw returns the next 32-bit integer word from the input *stream*. It returns the constant **EOF** on end-of-file or error, but since that is a good integer value, **feof** and **ferror**(3S) should be used to check the success of **getw**. **Getw** does not assume any special alignment in the file.

NOTES

The EOF return from **getchar** is incompatible with that used in early versions (1-6) of the UNIX System.

Because it is implemented as a macro, getc treats a *stream* argument with side effects incorrectly. Specifically, "getc(*f++);" doesn't work the way you might expect.

DIAGNOSTICS

These functions return the integer constant **EOF** on 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**(3S).

RELATED INFORMATION

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

NAME

gets, fgets – get a string from a stream

USAGE

#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 ends with a newline character, which is replaced in *s* by a null character. Gets returns its argument.

Fgets reads at most *n* -1 characters from *stream* into the string *s*. It stops at the first newline character, even if *n* characters have not yet been read. The last character read into *s* is followed by a null character. **Fgets** returns its first argument.

NOTES

Gets deletes a newline from the string it reads; **fgets** keeps it.

DIAGNOSTICS

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

RELATED INFORMATION

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

NAME

printf, fprintf, sprintf – formatted output conversion

USAGE

#include <stdio.h>

printf(*format [, arg] ...* **)**
**char ****format*;

fprintf(*stream, format [, arg] ...* **)**
**FILE ****stream*;
**char ****format*;

sprintf(*s, format [, arg] ...* **)**
**char ****s*, **format*;

DESCRIPTION

These functions write formatted output on a string or stream. **Printf** writes its output on the standard output stream **stdout**. **Fprintf** writes its output on the named output *stream*. **Sprintf** writes its "output," followed by a NULL character, into the string *s*.

The *format* argument to each of these functions controls conversion, format, and printing of the remaining arguments. *Format* is a character string that contains ordinary characters and conversion specifiers. The ordinary characters are simply copied to the output. Each conversion character is introduced by a % sign, and controls conversion and printing of an *arg*.

The first conversion specifier affects the first *arg*. The second conversion specifier affects the second *arg*, and so on through an arbitrary number of conversion specifiers and *arg*s

Following the %, a conversion specifier may include:

- 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, the value will be padded with zero instead of blanks. A field width may be specified by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width.
- An optional period (.), which serves to separate the field width from the next digit string.

- An optional digit string specifying a precision (number of digits to appear after the decimal point) for e- and f-conversion, or the maximum number of characters to be printed from a string. A precision may also be specified as an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width.
- an optional pound sign (#) specifying that the value should be converted to an "alternate form." This option has no effect on **c**, **d**, **s**, and **u** conversions. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero. For x(X) conversion, a non-zero result has the string **0**x(**0**X) prepended to it. For **e**, E, **f**, **g**, and **G**, conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point only appears in the results of those conversions if a digit follows the decimal point). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be.
- The character **l**, which specifies that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg*.
- One of the following characters, which indicates the type of conversion to be applied.

  d The integer *arg* is converted to decimal notation.

  o The integer *arg* is converted to octal notation.

  x The integer *arg* is converted to hexadecimal notation.

  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, six digits are given; if the precision is explicitly zero, 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, six digits are produced.

  g The float or double *arg* is printed in style **d**, in style **f**, or in style **e**, whichever gives full precision in minimum space.

  c The character *arg* is printed.

  s *Arg* is taken to be a string (character pointer) and characters from the string are printed until a null character is encountered or until the number of characters indicated by the precision specification is reached; however if the precision is zero 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 zero through 4294967295, the maximum value of an unsigned **int**.

% Print a percent sign (%); 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**(3S).

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

RELATED INFORMATION

putc(3S), scanf(3S), ecvt(3)

NAME

putc, putchar, fputc, putw – put character or word on a stream

USAGE

#include <stdio.h>

int putc(*c*, *stream***)**
char *c***;**
FILE **stream***;**

putchar(*c***)**

fputc(*c*, *stream***)**
char *c***;**
FILE **stream***;**

putw(*w*, *stream***)**
FILE **stream***;**

DESCRIPTION

The macro **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 function rather than a macro.

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.

NOTES

Because it is implemented as a macro, **putc** treats a *stream* argument with side effects improperly. In particular, "**putc(c, *f++);**" doesn't work correctly.

An error generated by a **putc** call can appear long after the erroneous call is executed.

DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, you must use **ferror**(3S) to detect **putw** errors.

RELATED INFORMATION

fopen(3S), **fclose**(3S), **getc**(3S), **puts**(3S), **printf**(3S), **fread**(3S)

NAME

puts, fputs – put a string on a stream

USAGE

#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.

RELATED INFORMATION

fopen(3S), gets(3S), putc(3S), printf(3S), ferror(3S), fread(3S)

NAME

scanf, fscanf, sscanf – formatted input conversion

USAGE

#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 the prescribed *format*, and stores the results in its arguments. Each expects as arguments a control string *format*, described below, and a set of *pointer* arguments that indicate 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:

- Blanks, tabs, or newlines, which match optional white space in the input.
- An ordinary character (not %) which must match the next character of the input stream.
- Conversion specifications, consisting of the percent character (%), an optional assignment-suppressing asterisk character (*), an optional numerical maximum field width, and a conversion character.

A conversion specification controls conversion of the next input field; the result is placed in the variable that the corresponding argument points to, unless assignment suppression, indicated by an asterisk (*), is specified. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

% a single % is expected in the input at this point; no assignment is done.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating "\0", which will be added. 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. The indicated number of characters is read.

e, f a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a **float**. The input format for floating point numbers is an optionally signed string of digits possibly containing a decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a 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 that 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**, with a preceding **h**, 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 zero, which means that no conversion was done. If conversion was intended, a return of zero means it did not take place due to an inappropriate character in the input.

EXAMPLES

The following call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

when presented with the following input line

```
25   54.32E-1  thompson
```

will assign the value 25 to **i**, the value 5.432 to **x**, and place the string "thompson\0" in **name**.

In another example, the call:

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

given the input data

```
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".

NOTES

The success of literal matches and suppressed assignments can not be determined directly.

DIAGNOSTICS

The **scanf** functions return **EOF** on end of input, and a short count for missing or illegal data items.

RELATED INFORMATION

atof(3), getc(3S), printf(3S)

NAME

setbuf, setbuffer, setlinebuf – assign buffering to a stream

USAGE

#include <stdio.h>

setbuf(*stream*, *buf*)
FILE **stream*;
char **buf*;

setbuffer(*stream*, *buf*, *size*)
FILE **stream*;
char **buf*;
int *size*;

setlinebuf(*stream*)
FILE **stream*;

DESCRIPTION

Three types of buffering are available: unbuffered, block-buffered, and line-buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block-buffered, many characters are saved up and written as a block; when it is line-buffered, characters are saved up until a newline is encountered or input is read from **stdin**. **Fflush** (see **fclose**(3S)) may be used to force the block out early. Normally, all files are block-buffered. A buffer is obtained from **malloc**(3) upon the first **getc**(3S) or **putc**(3S) call on a file. If the standard stream **stdout** refers to a terminal, the output is line-buffered. The standard stream **stderr** is always unbuffered.

Setbuf is used after a stream has been opened, but before it is read or written. The character array **buf** is 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, as shown here.

```
char buf[BUFSIZ];
```

Setbuffer, an alternate form of **setbuf**, is used after a stream has been opened, but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant pointer **NULL**, input/output will be completely unbuffered.

Setlinebuf is used to change **stdout** or **stderr** from block-buffered or unbuffered to line-buffered. Unlike **setbuf** and **setbuffer**, it can be used at any time that the file descriptor is active.

A file can be changed from unbuffered or line-buffered to block-buffered by using **freopen** (see **fopen**(3S)). A file can be changed from block-buffered or line-buffered to unbuffered by using **freopen** followed by **setbuf** with a buffer argument of **NULL**.

RELATED INFORMATION

fopen(3S), **getc**(3S), **putc**(3S), **malloc**(3), **fclose**(3S), **puts**(3S), **printf**(3S), **fread**(3S)

NAME

ungetc – push character back into input stream

USAGE

#include <stdio.h>

ungetc(*c*, *stream***)**
FILE **stream*;

DESCRIPTION

Ungetc pushes the character *c* back into the named 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 that something has been read from the stream and the stream is actually buffered. Attempts to **putc** an EOF are rejected.

Fseek(3S) erases all memory of pushed-back characters.

DIAGNOSTICS

Ungetc returns EOF if it can't push a character back onto the named *stream*.

RELATED INFORMATION

getc(3S), **setbuf**(3S), **fseek**(3S)

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

USAGE

#include <stdio.h>
#include <varargs.h>

int vprintf(*format, ap***)**
char **format*;
va_list *ap*;

int vfprintf(*stream*, *format*, *ap***)**
FILE **stream*;
char **format*;
va_list *ap*;

int vsprintf(*s*, *format*, *ap***)**
char **s*, **format*;
va_list *ap*;

DESCRIPTION

Vprintf, vfprintf, and **vsprintf** are analogous to **printf**(3S), **fprintf**(3S), and **sprintf**(3S) respectively, with one exception. Instead of being called with a variable number of arguments, they are called with an argument list as defined by **varargs**(5).

EXAMPLE

The example on the next page demonstrates how **vfprintf** could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 *
 */
/* VARARGS0 */
void
error(va_alist)
/*      Note that the function_name and format arguments cannot be
 *      separately declared because of the definition of varargs.
 */
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /* print out remainder of message */
        (void) vfprintf(fmt, args);
        va_end(args);
        (void)abort( );
}
```

RELATED INFORMATION

vprintf(3X), **varargs**(5).

NAME

intro – introduction to miscellaneous library functions

DESCRIPTION

These functions constitute minor libraries and other miscellaneous run-time facilities. They include device-independent plotting functions, terminal-independent screen management routines for two dimensional non-bitmap display terminals, functions for managing databases with inverted indexes, and sundry routines used in executing commands on remote machines.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
assert	assert.3x	program verification
curses	curses.3x	screen functions with ''optimal'' cursor motion
dbminit	dbm.3x	database subroutines
delete	dbm.3x	database subroutines
fetch	dbm.3x	database subroutines
firstkey	dbm.3x	database subroutines
initgroups	initgroups.3x	initialize group access list
nextkey	dbm.3x	database subroutines
rcmd	rcmd.3x	routines for returning a stream to a remote command
rexec	rexec.3x	return stream to a remote command
rresvport	rcmd.3x	routines for returning a stream to a remote command
ruserok	rcmd.3x	routines for returning a stream to a remote command
store	dbm.3x	database subroutines
tgetent	termcap.3x	terminal independent operation routines
tgetflag	termcap.3x	terminal independent operation routines
tgetnum	termcap.3x	terminal independent operation routines
tgetstr	termcap.3x	terminal independent operation routines
tgoto	termcap.3x	terminal independent operation routines
tputs	termcap.3x	terminal independent operation routines

NAME

assert – program verification

USAGE

#include <sdtio.h>
#include <assert.h>

assert(*expression***)**

DESCRIPTION

Assert is a macro that indicates that *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 (zero). Compiling with the **cc**(1), option **–DNDEBUG** effectively deletes **assert** from the program.

DIAGNOSTICS

"Assertion failed: file *f* line *n*". *F* is the name of the source file, and *n* is the line number of the **assert** statement in the source file.

NAME

curses – screen functions with optimized cursor motion

USAGE

cc *[flags] files* **-lcurses -ltermcap** *[libraries]*

DESCRIPTION

These routines provide a means of updating screens of dumb (and not-so-dumb) terminals in a reasonably optimal way. The routines keep an image of the current screen, and you set up an image of a new one. Then the **refresh( )** tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine **initscr( )** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin( )** should be called before exiting.

FUNCTIONS

addch(*ch* **)**	add a character to *stdscr*
addstr(*str* **)**	add a string to *stdscr*
box(*win,vert,hor* **)**	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(*scr,boolf* **)**	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delch()	delete a character
deleteln()	delete a line
delwin(*win* **)**	delete *win*
echo()	set echo mode
endwin()	end window modes
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getcap(*name* **)**	get terminal capability *name*
getstr(*str* **)**	get a string through *stdscr*
gettmode()	get tty modes
getyx(*win,y,x* **)**	get (y,x) coordinates
inch()	get char at current (y,x) coordinates
initscr()	initialize screens
insch(*c* **)**	insert a char
insertln()	insert a line
leaveok(*win,boolf* **)**	set leave flag for *win*
longname(*termbuf,name* **)**	get long name from *termbuf*
move(*y,x* **)**	move to (y,x) on *stdscr*
mvcur(*lasty,lastx,newy,newx* **)**	actually move cursor
newwin(*lines,cols,begin_y,begin_x* **)**	create a new window
nl()	set newline mapping

nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode
overlay(*win1,win2* **)**	overlay win1 on win2
overwrite(*win1,win2* **)**	overwrite win1 on top of win2
printw(*fmt,arg1,arg2,...* **)**	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
resetty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(*fmt,arg1,arg2,...* **)**	scanf through *stdscr*
scroll(*win* **)**	scroll *win* one line
scrollok(*win,boolf* **)**	set scroll flag
setterm(name)	set term variables for name
standend()	end standout mode
standout()	start standout mode
subwin(*win,lines,cols,begin_y,begin_x* **)**	create a subwindow
touchwin(*win* **)**	"change" all of *win*
unctrl(*ch* **)**	printable version of *ch*
waddch(*win,ch* **)**	add char to *win*
waddstr(*win,str* **)**	add string to *win*
wclear(*win* **)**	clear *win*
wclrtobot(*win* **)**	clear to bottom of *win*
wclrtoeol(*win* **)**	clear to end of line on *win*
wdelch(*win,c* **)**	delete char from *win*
wdeleteln(*win* **)**	delete line from *win*
werase(*win* **)**	erase *win*
wgetch(*win* **)**	get a char through *win*
wgetstr(*win,str* **)**	get a string through *win*
winch(*win* **)**	get char at current (y,x) in *win*
winsch(*win,c* **)**	insert char into *win*
winsertln(*win* **)**	insert line into *win*
wmove(*win,y,x* **)**	set current (y,x) coordinates on *win*
wprintw(*win,fmt,arg1,arg2,...* **)**	printf on *win*
wrefresh(*win* **)**	make screen look like *win*
wscanw(win, *fmt,arg1,arg2,...* **)**	scanf through *win*
wstandend(*win* **)**	end standout mode on *win*
wstandout(*win* **)**	start standout mode on *win*

RELATED INFORMATION

DOMAIN/IX Support Tools Guide
ioctl(2), **getenv**(3), **tty**(4)

NAME

dbminit, fetch, store, delete, firstkey, nextkey – database subroutines

USAGE

```
typedef struct {
        char *dptr;
        int dsize;
} datum;

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;
```

DESCRIPTION

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will find a keyed item in one or two file system accesses. You must link with *libdbm.a*, using the loader option **-ldbm**, to access these functions.

The **datum** typedef decsribes the **keys** and **contents**. A **datum** specifies a string of **dsize** bytes pointed to by **dptr.** Both arbitrary binary data and normal ASCII strings are allowed. The database is stored in two files. One file is a directory containing a bit map and has ".dir" as its suffix. The second file contains all data and has ".pag" as its suffix.

Before you can access a database, you must open it with **dbminit.** At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length ".dir" and ".pag" files.)

Once open, **fetch** accesses data stored under a key; **store** places data under a key. **Delete** removes a key (and its associated contents). A linear pass through all keys in a database may be made, in an (apparently) random order, by use of **firstkey** and **nextkey**. **Firstkey** will return the first key in the database. With any key, **nextkey** will return the next key in the database.

EXAMPLE

This code will traverse the database:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

FILES

libdbm.a library of database routines

NOTES

The ".pag" file will contain holes; its apparent size is about four times larger than its content. These files cannot be copied by normal means (**cp**, **cat**, **tp**, **tar**, **ar**) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that subsequent calls change.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover, all key/content pairs that hash together must fit on a single block. **Store** will return an error if a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys that **firstkey** and **nextkey** present depends on a hashing function.

DIAGNOSTICS

All functions that return an **int** on success indicate errors with negative values. A zero return indicates that the function was successful. Routines that return a **datum** indicate errors with a null (0) **dptr.**

NAME

initgroups – initialize group access list

USAGE

initgroups(*name*, *basegid*)
char **name*;
int *basegid*;

DESCRIPTION

Initgroups reads through the group file and sets up, using the **setgroups**(2) call, the group access list for the user specified in *name*. The *basegid* is included automatically in the groups list. Typically, this value is the group number from the password file.

NOTES

Initgroups uses the routines based on **getgrent**(3). If the invoking program uses any of these routines, the group structure will be overwritten in the call to **initgroups**.

The */etc/group* file must be kept up-to-date. On DOMAIN/IX Systems, the program /etc/**crpasswd** handles this chore.

DIAGNOSTICS

Initgroups returns -1 if the process is not super-user.

FILES

/etc/group	the group file

RELATED INFORMATION

setgroups(2), **crpasswd**(8)

NAME

openpl, erase, label, line, circle, arc, move, cont, point, linemod, space, closepl – graphics interface

USAGE

openpl()

erase()

label(*s*)
char *s*[ ];

line(*x1*, *y1*, *x2*, *y2*)

circle(*x*, *y*, *r*)

arc(*x*, *y*, *x0*, *y0*, *x1*, *y1*)

move(*x*, *y*)

cont(*x*, *y*)

point(*x*, *y*)

linemod(*s*)
char *s*[ ];

space(*x0*, *y0*, *x1*, *y1*)

closepl( )

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot*(5) for a description of their effect. **Openpl** must be used before any of the others to open the device for writing. **Closepl** flushes the output.

String arguments to **label** and **linemod** are null-terminated and do not contain newlines.

Various flavors of these functions exist for different output devices. They are obtained by the following **ld**(1) options:

-lplot produce a device-independent graphics stream on standard output for **plot**(1) filters

-lgmr produce a DOMAIN 2D Graphics Metafile Resource (2DGMR) file.

FILES

libplot.a library of plotting functions

RELATED INFORMATION

plot(5), plot(1G), graph(1G)

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

USAGE

rem = rcmd(*ahost, inport, locuser, remuser, cmd, fd2p* **);**
char ***ahost*;
u_short *inport*;
char **locuser*, **remuser*, **cmd*;
int **fd2p*;

s = **rresvport(** *port* **);**
int **port*;

ruserok(*rhost, superuser, ruser, luser* **);**
char **rhost*;
int *superuser*;
char **ruser*, **luser*;

DESCRIPTION

Rcmd is used by the super-user to execute a command on a remote machine using a dubious authentication scheme based on reserved port numbers. **Rresvport** returns a descriptor to a socket with an address in the privileged port space. **Ruserok** is used by servers to authenticate clients requesting service with **rcmd**. All three functions are present in the same file and are used by the **rshd**(8) server (among others).

Rcmd looks up the host **ahost* using **gethostbyname**(3N). It returns -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, which it forwards to the process group of the command. If *fd2p* is zero, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision will be made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in **rshd**(1M).

The **rresvport** routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by **rcmd** and several other routines. Privileged addresses consist of a port in the range zero to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a **gethostent**(3N) routine, two usernames and a flag indicating if the local username is the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A 1 is returned if the machine name is listed in *hosts.equiv*, or the host and remote username are found in the **ruserok** returns zero. If the *superuser* flag is 1, the check of *host.equiv* is bypassed.

NOTES

There is no way to specify options to the **socket** call that **rcmd** makes.

RELATED INFORMATION

rlogin(1), **rsh**(1), **rexec**(3X), **rexecd**(8), **rlogind**(8), **rshd**(8)

NAME

rexec – return stream to a remote command

USAGE

rem = rexec(*ahost*, *inport*, *user*, *passwd*, *cmd*, *fd2p*);
char ***ahost*;
u_short *inport*;
char **user*, **passwd*, **cmd*;
int **fd2p*;

DESCRIPTION

Rexec looks up the host **ahost* using **gethostbyname**(3N). It returns -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a user-name and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in the user's home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

Inport specifies which well-known DARPA Internet port to use for the connection; it will normally be the value returned from the call

getservbyname(exec, tcp)

(see **getservent**(3N)). The protocol for connection is described in detail in **rexecd**(8).

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then a auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being signal numbers to be forwarded to the process group of the command. If *fd2p* is zero, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout**, and no provision will be made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

NOTES

There is no way to specify options to the **socket** call that **rexec** makes.

RELATED INFORMATION

rcmd(3X), **rexecdm**(8)

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operation routines

USAGE

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 entries from the terminal capability database */etc/termcap*, described in **termcap**(5). These are low level routines; for a higher-level package, see **curses**(3X).

Tgetent extracts the entry for terminal *name* and puts it into the buffer pointed to by *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, zero 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 it finds one, and the value does not begin with a slash, and the terminal type *name* is the same as the environment string TERM, it reads the TERMCAP string instead of *termcap* file. If it does begin with a slash, it assumes the string is a pathname to be used instead of */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 on */etc/termcap*.

Tgetnum gets the numeric value of entry *id,* returning -1 if it is not given for the terminal. **Tgetflag** returns 1 if the specified capability is present in the terminal's entry, zero if it is not. **Tgetstr** gets the string value of capability *id,* placing it in the buffer at *area,* and advancing the *area* pointer. It decodes all abbreviations for this field described in **termcap**(5) except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline.* It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D**, or **^@** in the returned string. (Programs that call **tgoto** should turn off the XTABS bit(s), since **tgoto** may now output a tab. Programs using **termcap** should, in general, turn off XTABS since some terminals use ↑I for other functions, such as nondestructive space.) If an incomprehensible % sequence is given, **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**(3). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	library of **termcap** routines.
/etc/termcap	terminal capabilities database

RELATED INFORMATION

ex(1), **vi**(1), **curses**(3X), **termcap**(5)

This is a topical index for Section 3 of the *DOMAIN/IX Programmer's Reference Manual for BSD4.2*. For a permuted index of all reference information, see Appendix A of this manual.

NAME

special files – introduction to special files

DESCRIPTION

This section describes various special files found in the */dev* directory. With a few exceptions, these files are devices or pseudo-devices, and reside in the directory */dev*. On DOMAIN Systems, */dev* is typically a link to *`node_data/dev*.

NAME

mtio – tape device files

DESCRIPTION

The files in */dev/r?t** refer to tape I/O devices. These files are created using the **/com/edmtdesc** (edit magtape descriptor) command.

The block length associated with */dev/rmt** files is 1024 bytes. Cartridge tape (*/dev/rct**) files have a block length of 512 bytes. If you need to change the block length (or change or examine any other parameter of a magtape descriptor file) use **/com/edmtdesc**.

FILES

Tape device filenames are:

/dev/rmt8	magtape, rewind on file close
/dev/rmt12	magtape, no rewind on file close
/dev/rct8	cartridge tape, rewind on file close
/dev/rct12	cartridge tape, no rewind on file close

NAME

null – data sink

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return zero bytes.

FILES

/dev/null

NAME

pty – pseudo terminal driver

USAGE

pseudo-device pty

DESCRIPTION

The **pty** driver provides support for a device-pair termed a *pseudo terminal*. A pseudo terminal is a pair of character devices, a "master" device and a "slave" device. The slave device provides processes with an interface identical to that described in **tty**(4). However, whereas all other devices which provide the interface described in **tty** have some hardware device behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal. That is, anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device.

On DOMAIN/IX Systems, the program /etc/**crpty** creates **pty** pairs (see **crpty**(8)). If invoked with no optional "count," 16 pseudo terminal pairs are configured.

The following **ioctl**(2) calls apply only to **ptys**:

TIOCSTOP Stops output to a terminal (e.g. like typing ^S). Takes no parameter.

TIOCSTART
Restarts output (stopped by TIOCSTOP or by typing ^S). Takes no parameter.

TIOCPKT Enable/disable packet mode. Packet mode is enabled by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. When applied to the master side of a pseudo terminal, each subsequent **read**(2) from the terminal will return data written on the slave part of the pseudo terminal preceded by a zero byte (symbolically defined as TIOCPKT_DATA), or a single byte reflecting control status information. In the latter case, the byte is an inclusive-or of zero or more of the bits:

TIOCPKT_FLUSHREAD
whenever the read queue for the terminal is flushed.

TIOCPKT_FLUSHWRITE
whenever the write queue for the terminal is flushed.

TIOCPKT_STOP whenever output to the terminal is stopped with a ^S.

TIOCPKT_START whenever output to the terminal is restarted.

TIOCPKT_DOSTOP	whenever **t_stopc** is ^S and **t_startc** is ^Q.
TIOCPKT_NOSTOP	whenever the start and stop characters are not ^S/^Q.

This mode is used by **rlogin**(1) and **rlogind**(8C) to implement a remote-echoed, locally ^S/^Q flow-controlled remote login with proper back-flushing of output; it can be used by other similar programs.

TIOCREMOTE A mode for the master half of a pseudo terminal, independent of TIOCPKT. This mode causes input to the pseudo terminal to be flow controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. In normal usage, a write of data is like the data typed as a line on the terminal; a write of zero bytes is like typing an end-of-file character. TIOCREMOTE can be used when doing remote line editing in a window manager, or whenever flow-controlled input is required.

FILES

/dev/pty[p-r][0-9a-f]	master pseudo terminals
/dev/tty[p-r][0-9a-f]	slave pseudo terminals

NAME

tty – general terminal interface

USAGE

#include <sgtty.h>

DESCRIPTION

This manual entry normally describes the special file */dev/tty*, as well as the system's terminal drivers. While DOMAIN Systems do not support */dev/tty* as such, DOMAIN/IX software supports a large subset of the UNIX System **tty** interface over SIO (Serial I/O) lines (*/dev/sio**), in **vt100** windows (DM windows controlled by the **/com/vt100** process), and over **pty**(4), or pseudo-tty, connections. However, it is probably most common for users to log in to the Display Manager (DM) and transact their business via a shell that echos standard input in an "input pad," writes output to a "transcript pad," and, in general, supports only a small subset of **tty** functionality.

In this entry, we describe the abstract **tty** interface. Entries for specific devices describe the subset of **tty** functionality that those devices support.

Note — Any applicable "default" key bindings mentioned in this entry can be put into effect for the DM by executing one of the */sys/dm/bsd4.2_keys?* key definitions files.

Line Disciplines

There are two "line disciplines" that affect the handling of **tty**'s:

old The old (standard) line discipline, used by **/bin/sh**, and where needed for compatibility with older (version 7) UNIX systems.

new A newer terminal driver, with features for job control required by the C Shell, **/bin/csh**.

Line discipline switching is accomplished with the TIOCSETD **ioctl**:

int ldisc = *LDISC***;**

ioctl(f, TIOCSETD, &ldisc);

where LDISC is OTTYDISC for the standard tty driver or NTTYDISC for the new driver. The standard (old) tty driver is discipline 0 by convention. The current line discipline can be obtained with the TIOCGETD **ioctl**. Pending input is discarded when the line discipline is changed.

All DOMAIN System serial communications ports can use either line discipline.

The Control Terminal

When a terminal file is opened, it causes the process to wait until a connection is established. These files are typically opened by the login process and become the user's standard input and output file.

If a process that has no control terminal opens a terminal file, then that terminal file becomes the control terminal for that process. The control terminal is thereafter inherited by a child process during a **fork**(2), even if the control terminal is closed.

The file */dev/tty* is, in each process, a synonym for the "control terminal" associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected, or when a program requires a handy file name for output.

A process can remove the association it has with its controlling terminal by opening the file */dev/tty* and issuing a

```
ioctl(f, TIOCNOTTY, 0)
```

This is often desirable in server processes.

Process Groups

Command processors such as **csh**(1) can arbitrate the terminal between different "jobs" by placing related jobs in a single process group and associating this process group with the terminal. A terminal's associated process group may be set using the TIOCSPGRP **ioctl**.

```
ioctl(fildes, TIOCSPGRP, &pgrp);
```

or examined using TIOCGPGRP, which returns the current process group in *pgrp*. The new terminal driver aids in this arbitration by restricting access to the terminal by processes which are not in the current process group; see **Job Access Control** below.

Modes

The terminal drivers have three major modes, characterized by the amount of processing on the input and output characters.

cooked The normal mode. In this mode lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when the **t_brkc** character, normally an EOT, is entered. A carriage return is usually made synonymous with newline in this mode, and is replaced with a newline whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, etc.) are available in this mode.

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

CBREAK This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next and interrupt processing are still done in this mode. Output processing is done.

The style of input processing can also be very different when the terminal is put in non-blocking I/O mode; see the FNDELAY flag as described in **fcntl**(2). In this case a **read**(2) from the control terminal will never block, but rather return an error indication (EWOULDBLOCK) if there is no input available.

A process may also request a SIGIO signal be sent it whenever input is present. To enable this mode the FASYNC flag should be set using **fcntl**.

Input Editing

A UNIX System terminal ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring. Input characters are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. In RAW mode, the terminal driver throws away all input and output without notice when the limit is reached. In CBREAK or cooked mode it refuses to accept any further input and, if in the new line discipline, sounds the terminal bell.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word it is possible to have input characters with that parity discarded (see the **Summary** below).

In all of the line disciplines, it is possible to simulate terminal input using the TIOCSTI **ioctl**, which takes as its third argument the address of a character. The system pretends that this character was typed on the argument terminal, which must be the control terminal unless the caller is the super-user

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the **stty**(3C) call or the TIOCSETN or TIOCSETP **ioctls** (see the **Summary** below).

In cooked mode, terminal input is processed in units of lines. A program attempting to read will normally be suspended until an entire line has been received (see the description of SIGTTIN in **Job access control** and of FIONREAD in **Summary**, both below). No matter how many characters are requested in the **read** call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters — even one — may be requested in a **read** without losing information.

During input, line editing is normally done, with the DELETE character (normally mapped to the <BACK SPACE> key) logically erasing the last character typed and the character ↑U logically erasing the entire current input line. These characters never erase beyond the beginning of the current input line or a ↑D. These characters may be entered literally by preceding them with '\'; the '\' will normally be erased when the character is typed.

The drivers normally treat either a carriage return or a newline character as terminating an input line, replacing the return with a newline and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word then the processing for carriage return is disabled, and it is simply echoed as a return, and does not terminate cooked mode input.

In the new driver there is a literal-next character, ↑V, which, in both cooked and CBREAK mode, removes any special meaning that would otherwise be attached to the character it immediately precedes. While use of ↑V is a preferable method of escaping erase and kill characters, '\' retains its old function in the new line discipline.

The new terminal driver also provides two other editing characters in normal mode. The word-erase character, normally ↑W, erases the preceding word, but not any spaces before it. For the purposes of ↑W, a word is defined as a sequence of non-blank characters, with tabs counted as blanks. Finally, the reprint character, normally ↑R, retypes the pending input beginning on a new line. Retyping occurs automatically in cooked mode if characters that would normally be erased from the screen are fouled by program output.

Input Echoing and Redisplay

The terminal driver has several modes for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy Terminals

When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters which are logically erased are then printed out backwards preceded by '\' and followed by '/' in this mode.

CRT Terminals

When a CRT terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal driver then echoes the proper number of erase characters when input is erased; in the normal case where the erase character is a ^H this causes the cursor of the terminal to back up to where it was before the logically erased character was typed. If the input has become fouled due to interspersed asynchronous output, the input is automatically retyped.

Erasing Characters from a CRT

When a CRT terminal is in use, the LCRTERA bit may be set to cause input to be erased from the screen with a "backspace-space-backspace" sequence when character or word deleting sequences are used. A LCRTKIL bit may be set as well, causing the input to be erased in this manner on line kill sequences as well.

Echoing of Control Characters

If the LCTLECH bit is set in the local state word, then non-printing (control) characters are normally echoed as ^X (for some X) rather than being echoed unmodified; delete is echoed as ^?.

The normal modes for use on CRT terminals are speed-dependent. At speeds less than 1200 baud, LCRTERA and LCRTKILL processing can be quite slow, so **stty** normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater all of these bits are normally set. **Stty** summarizes these option settings and the use of the new terminal driver as "newcrt."

Output Processing

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is normally generated on output. The EOT character is not transmitted in cooked mode to prevent terminals that respond to it from hanging up; programs using RAW or CBREAK mode should be careful.

The terminal drivers provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces ↑H, form feeds ↑L, carriage returns ↑M, tabs ↑I and newlines ↑J. The driver will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns, and optionally convert newlines to carriage returns followed by newline. These functions are controlled by bits in the tty flags word; see the **Summary** below.

The terminal drivers provide for mapping between upper and lower case on terminals lacking lower case, and for other special processing on deficient terminals.

Finally, in the new terminal driver, there is a output flush character, normally ^O, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. An **ioctl** to flush the characters in the input or output queues, TIOCFLUSH, is also available.

Uppercase-Only Terminals and Hazeltines

If the LCASE bit is set in the tty flags, then all upper-case letters are mapped into the corresponding lower-case letter. To generate an uppercase letter, precede it by '\'. Upper case letters are preceded by a '\' when output. In addition, the following escape sequences can be generated on output and accepted on input:

Character	Escape Sequence
`	\\'
|	\\!
~	\\^
{	\\(
}	\\)

To deal with Hazeltine terminals, which do not understand that tilde (~) has been made into an ASCII character, the LTILDE bit may be set in the local mode word; in this case the character ~ will be replaced with the character ` on output.

Flow Control

There are two characters (the stop character, normally ↑S, and the start character, normally ↑Q) which cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode the system produces a stop character (default ↑S) when the input queue is in danger of overflowing, and a start character (default ↑Q) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys the conventions.

Line Control and Breaks

There are several **ioctl** calls available to control the state of the terminal line. The TIOCSBRK **ioctl** will set the break bit in the hardware interface causing a break condition to exist; this can be cleared (usually after a delay with **sleep**(3)) by TIOCCBRK. Break conditions in the input are reflected as a null character in RAW mode or as the interrupt character in cooked or CBREAK mode. The TIOCCDTR **ioctl** will clear the data terminal ready condition; it can be set again by TIOCSDTR.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a SIGHUP hangup signal is sent to the processes in the distinguished process group of the terminal; this usually causes them to terminate (the SIGHUP can be suppressed by setting the LNOHANG bit in the local state word of the driver.) Access to the terminal by other processes is then normally revoked, so any further **reads** will fail, and programs that read a terminal and test for End-Of-File on their input will terminate appropriately.

When using an ACU it is possible to ask that the phone line be hung up on the last close with the TIOCHPCL **ioctl**; this is normally done on the outgoing line.

Interrupt Characters

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent to the processes in the control group of the terminal, as if a TIOCGPGRP **ioctl** were done to get the process group and then a **killpg**(2) system call were done, except that these characters also flush pending input and output when typed at a terminal (*à la* TIOCFLUSH). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed..TP 1i **Note** Any applicable "default" key bindings mentioned in this entry can be put into effect for the DM by executing one of the */sys/dm/bsd4.2_keys?* key definitions files.

↑C **t_intrc** (ETX) generates a SIGINT signal. This is the normal way to stop a process which is no longer interesting, or to regain control in an interactive program.

↑\ **t_quitc** (FS) generates a SIGQUIT signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.

↑Z **t_suspc** (EM) generates a SIGTSTP signal, which is used to suspend the current process group.

↑Y **t_dsuspc** (SUB) generates a SIGTSTP signal as ↑Z does, but the signal is sent when a program attempts to read the ↑Y, rather than when it is typed.

Job Access Control

When using the new terminal driver, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal its process group is sent a SIGTTIN signal. This signal normally causes the members of that process group to stop. If, however, the process is ignoring SIGTTIN, has SIGTTIN blocked, or is in the middle of process creation using **vfork**(2)), the read will return –1 and set *errno* to *EIO*.

When using the new terminal driver with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal if it is not in the distinguished process group for that terminal. Processes which are holding or ignoring SIGTTOU signals or which are in the middle of a **vfork** are excepted and allowed to produce output.

Summary of Modes

There are 4 different structures which contain various portions of the driver data. (This is an unfortunate side effect of the evolution of the tty driver.) The first of these (*sgttyb*) contains that part of the information largely common between version 6 and version 7 UNIX systems. The second contains additional control characters added in version 7. The third is a word of local state added in 4BSD, and the fourth is another structure of special characters added for the new driver. In the future a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty

The basic **ioctl**s use the structure defined in <*sgtty.h*>:

```
struct sgttyb {
        char    sg_ispeed;
        char    sg_ospeed;
        char    sg_erase;
        char    sg_kill;
        short   sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <*sgtty.h*>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are <BACK SPACE> and ↑U.)

The *sg_flags* field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input and generated on output
ODDP	0000100	Odd parity allowed on input and generated on output
RAW	0000040	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0000020	Map CR into LF; output LF as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input and lower to upper on output
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for exceptionally slow terminals.

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

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is suitable for the concept-100 and pads lines to be at least 9 characters at 9600 baud.

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

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Input characters with the wrong parity, as determined by bits 200 and 100, are ignored in cooked and CBREAK mode.

RAW disables all processing except output flushing with LFLUSHO; full 8 bits of input are given as soon as it is available; all 8 bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode all data in the input and output queues are discarded; this applies to both new and old drivers.

CRMOD causes input carriage returns to be turned into new-lines, and output and echoed new-lines to be output as a carriage return followed by a line feed.

In CBREAK mode, programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ and EOT are disabled.

TANDEM mode causes the system to produce a stop character (default ^S) whenever the input queue is in danger of overflowing, and a start character (default ↑Q) when the input queue has drained sufficiently. It is useful for flow control when the "terminal" is really another computer which understands the conventions.

Note: The same "stop" and "start" characters are used for both directions of flow control; the *t_stopc* character is accepted on input as the character that stops output and is produced on output as the character to stop input, and the *t_startc* character is accepted on input as the character that restarts output and is produced on output as the character to restart input.

Basic ioctls

A large number of **ioctl** calls apply to terminals. Some have the general form:

#include <sgtty.h>

ioctl(*fildes*, *code*, *arg*)

struct sgttyb *arg;

The applicable *codes* are:

TIOCGETP Fetch the basic parameters associated with the terminal, and store in the pointed-to *sgttyb* structure.

TIOCSETP Set the parameters according to the pointed-to *sgttyb* structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN Set the parameters like TIOCSETP but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes *arg* is ignored.

TIOCEXCL
Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL
Turn off "exclusive-use" mode.

TIOCHPCL
When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

With the following codes *arg* is a pointer to an **int**.

TIOCGETD
arg is a pointer to an **int** into which is placed the current line discipline number.

TIOCSETD
arg is a pointer to an **int** whose value becomes the current line discipline number.

TIOCFLUSH
If the *int* pointed to by *arg* has a zero value, all characters waiting in input or output queues are flushed. Otherwise, the value of the *int* is treated as the logical OR of the FREAD and FWRITE defined in <sys/file.h>; if the FREAD bit is set, all characters waiting in input queues are flushed, and if the FWRITE bit is set, all characters waiting in output queues are flushed.

For the remaining calls, the arguments, where required, are described; *arg* should otherwise be given as 0.

TIOCSTI
the argument points to a character which the system pretends had been typed on the terminal. (Not supported on DOMAIN/IX.)

TIOCSBRK
the break bit is set in the terminal.

TIOCCBRK
the break bit is cleared.

TIOCSDTR
data terminal ready is set.

TIOCCDTR
data terminal ready is cleared.

TIOCSTOP
output is stopped as if the "stop" character had been typed.

TIOCSTART
output is restarted as if the "start" character had been typed.

TIOCGPGRP
arg is a pointer to an **int** into which is placed the process group ID of the process group for which this terminal is the control terminal.

TIOCSPGRP
arg is a pointer to an **int** (typically a process ID); the process group whose process group ID is the value of this **int** becomes the process group for which this terminal is the control terminal.

TIOCOUTQ
returns in the **int** pointed to by *arg* the number of characters queued up to be output to the terminal.

FIONREAD
returns in the **int** pointed to by *arg* the number of immediately readable characters from the argument unit. This works for files, pipes, and terminals.

Tchars

The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces: The following structure is defined in <*sys/ioctl.h*>, which is automatically included in <*sgtty.h*>:

```
struct tchars {
        char   t_intrc;      /* interrupt */
        char   t_quitc;      /* quit */
        char   t_startc;     /* start output */
        char   t_stopc;      /* stop output */
        char   t_eofc;       /* end-of-file */
        char   t_brkc;       /* input delimiter (like nl) */
};
```

The default values for these characters are

t_intrc (interrupt)	**↑?**
t_quit (quit)	**↑**
t_startc (start output)	**↑Q**
t_stopc (stop output)	**↑S**
t_eofc (end-of-file)	**↑D**
t_brkc (input delimiter)	**–1**

A character value of –1 eliminates the effect of that character. The *t_brkc* character, by default –1, acts like a new-line in that it terminates a "line," is echoed, and is passed to the program. The "stop" and "start" characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase and kill) identical. The applicable **ioctl** calls are:

TIOCGETC Get the special characters and put them in the specified structure.

TIOCSETC Set the special characters to those given in the structure.

Local Mode

The third structure associated with each terminal is a local mode word. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as backspace-space-backspace
LTILDE	000010	Convert ˜ to ` on output (for Hazeltine terminals)

LMDMBUF	000020	Stop/start output when carrier drops
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
LETXACK	001000	Diablo style buffer hacking (unimplemented)
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only ↑Q restarts output after ↑S, like DEC systems
LNOFLSH	100000	Inhibit flushing of pending I/O when an interrupt character is typed.

The applicable **ioctl** functions are:

TIOCLBIS *arg* is a pointer to an **int** whose value is a mask containing the bits to be set in the local mode word.

TIOCLBIC *arg* is a pointer to an **int** whose value is a mask containing the bits to be cleared in the local mode word.

TIOCLSET *arg* is a pointer to an **int** whose value is stored in the local mode word.

TIOCLGET *arg* is a pointer to an **int** into which the current local mode word is placed.

Local Special Chars

The final structure associated with each terminal is the *ltchars* structure which defines control characters for the new terminal driver. Its structure is:

```
struct ltchars {
        char    t_suspc;        /* stop process signal */
        char    t_dsuspc;       /* delayed stop process signal */
        char    t_rprntc;       /* reprint line */
        char    t_flushc;       /* flush output (toggles) */
        char    t_werasc;       /* word erase */
        char    t_lnextc;       /* literal next character */
};
```

The default values for these characters are:

t_suspc (stop)	↑Z
t_dsuspc (delayed stop)	↑Y

t_rprntc (reprint line)	↑**R**
t_flushc (flush output)	↑**O**
t_werasc (word erase)	↑**W**
t_lnextc (literal-next)	↑**V**

A value of –1 disables the character.

The applicable *ioctl* functions are:

TIOCSLTC *arg* is a pointer to an *ltchars* structure which defines the new local special characters.

TIOCGLTC *arg* is a pointer to an *ltchars* structure into which is placed the current set of local special characters.

FILES

/dev/tty	not supported on DOMAIN Systems
*/dev/tty**	links to */dev/sio**
/dev/console	not supported on DOMAIN Systems

RELATED INFORMATION

csh(1), **stty**(1), **ioctl**(2), **sigvec**(2), **stty**(3C), **getty**(8).

NAME

networking – introduction to networking facilities

USAGE

#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>

DESCRIPTION

This section briefly describes the 4.2BSD networking facilities available in the *bsd4.2* version of DOMAIN/IX. Documentation in this part of section 4 is broken up into three areas: protocol families, protocols, and network interfaces.

Entries describing a protocol family are marked (4F), while entries describing protocol use are marked (4P). Hardware support for network interfaces are found among the standard (4) entries.

All network protocols are associated with a specific protocol family. A protocol family provides the basic services a protocol implementation needs in order to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, though the current protocol implementations do not. A protocol-family is normally comprised of a number of protocols, one per **socket**(2) type. It is not required that a protocol-family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in **socket**(2). A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction may allow more than one byte of out-of-band data to be transmitted per out-of-band message.

A network interface is similar to a device interface. Network interfaces comprise the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families, and/or address formats. The USAGE section of each network interface entry gives a sample specification of the related drivers for use in providing a system description. The DIAGNOSTICS section lists various diagnostic messages generated by errors in device operation.

PROTOCOLS

DOMAIN/IX currently supports only the DARPA Internet protocols fully.

ADDRESSING

Associated with each protocol family is an address format. The following address formats are used by the system:

```
#define AF_UNIX      1   /* local to host (pipes, portals) */
#define AF_INET      2   /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK   3   /* arpanet imp addresses */
#define AF_PUP       4   /* pup protocols: e.g. BSP */
```

ROUTING

The network facilities provided limited packet routing. A simple set of data structures comprise a "routing table" used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this database with the aid of two socket-specific **ioctl**(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in *<net/route.h>*;

```
struct rtentry {
        u_long   rt_hash;
        struct   sockaddr rt_dst;
        struct   sockaddr rt_gateway;
        short    rt_flags;
        short    rt_refcnt;
        u_long   rt_use;
        struct   ifnet *rt_ifp;
};
```

with **rt_flags** defined from,

```
#define RTF_UP        0x1   /* route usable */
#define RTF_GATEWAY   0x2   /* destination is a gateway */
#define RTF_HOST      0x4   /* host entry (net otherwise) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted, each network interface autoconfigured installs a routing table entry when it wishes to have packets sent through it. Normally the interface specifies the route through it is a "direct" connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface may be requested to address the packet to an entity different from the eventual recipient (i.e., the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (**rt_refcnt** is non-zero), the resources associated with it will not be reclaimed until further references to it are released.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent entry, or ENOBUFS if insufficient resources were available to install a new route.

The **rt_use** field contains the number of packets sent along the route. This value is used to select among multiple routes to the same destination. When multiple routes to the same destination exist, the least used route is selected.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

INTERFACES

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces do not.

At boot time each interface which has underlying hardware support makes itself known to the system during the autoconfiguration process. Once the interface has acquired its address it is expected to install a routing table entry so that messages may be routed through it. Most interfaces require some part of their address specified with an SIOCSIFADDR ioctl before they will allow traffic to flow through them. On interfaces where the network-link layer address mapping is static, only the network number is taken from the ioctl; the remainder is found in a hardware specific manner. On interfaces which provide dynamic network-link layer address mapping facilities (e.g., 10Mb/s Ethernets), the entire address specified in the **ioctl** is used.

The following **ioctl** calls may be used to manipulate network interfaces. Unless specified otherwise, the request takes an **ifrequest** structure as its parameter. This structure has the form

```
struct  ifreq {
        char   ifr_name[16];          /* name of interface (e.g. "ec0") */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                short   ifru_flags;
        } ifr_ifru;
#define ifr_addr        ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_flags       ifr_ifru.ifru_flags     /* flags */
};
```

SIOCSIFADDR Set interface address. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR Get interface address.

SIOCSIFDSTADDR Set point to point address for interface.

SIOCGIFDSTADDR Get point to point address for interface.

SIOCSIFFLAGS Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS Get interface flags.

SIOCGIFCONF Get interface configuration list. This request takes an **ifconf** structure (see below) as a value-result parameter. The **ifc_len** field should be initially set to the size of the buffer pointed to by **ifc_buf**. On return it will contain the length, in bytes, of the configuration list.

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
```

```
struct  ifconf {
        int ifc_len;            /* size of associated buffer */
        union {
                caddr_t         ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf       /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req       /* array of structures returned */
};
```

RELATED INFORMATION

socket(2), **ioctl**(2), **intro**(4), **routed**(8)

NAME

inet – Internet protocol family

USAGE

#include <sys/types.h>
#include <netinet/in.h>

DESCRIPTION

The Internet protocol family is a collection of protocols layered atop the Internet Protocol (IP) transport layer, and utilizing the Internet address format. The Internet family provides protocol support for the SOCK_STREAM and SOCK_DGRAM socket types.

ADDRESSING

Internet addresses are four-byte quantities, stored in network standard format. The include file **<netinet/in.h>** defines this address as a discriminated union.

Sockets bound to the Internet protocol family utilize the following addressing structure,

```
struct sockaddr_in {
        short   sin_family;
        u_short         sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Sockets may be created with the address INADDR_ANY to effect "wildcard" matching on incoming messages.

PROTOCOLS

The Internet protocol family is comprised of the IP transport protocol, Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction while UDP is used to support the SOCK_DGRAM abstraction. The ICMP message and IP protocols are not directly accessible.

RELATED INFORMATION

tcp(4P), udp(4P), ip(4P)

NAME

arp – Address Resolution Protocol

DESCRIPTION

Arp is a protocol used to dynamically map between DARPA Internet addresses and addresses on the local network.

Arp caches Internet-local net address mappings. When an interface requests a mapping for an address not in the cache, **arp** queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending messages are transmitted. **Arp** will queue at most one packet while waiting for a mapping request to be responded to; only the most recently "transmitted" packet is kept.

NAME

tcp – Internet Transmission Control Protocol

USAGE

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket( AF_INET, SOCK_STREAM, 0);
```

DESCRIPTION

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses". Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilizing the tcp protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the **listen**(2) system call must be used after binding the socket with the **bind**(2) system call. Only passive sockets may use the **accept**(2) call to accept incoming connections. Only active sockets may use the **connect**(2) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, termed "wildcard addressing", allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	An attempt was made to establish a connection on a connected socket.
[ENOBUFS]	The system doesn't have enough memory to hold an internal data structure;
[ETIMEDOUT]	A connection was dropped after many retransmissions;
[ECONNRESET]	The remote peer forced the connection to be closed;

[ECONNREFUSED] The remote peer actively refused connection establishment (usually because no process is listening to the port).

[EADDRINUSE] An attempt was made to create a socket with a port that has already been allocated.

[EADDRNOTAVAIL] An attempt is made to create a socket with a network address for which no network interface exists.

RELATED INFORMATION

intro(4N), inet(4F)

NAME

udp – Internet User Datagram Protocol

USAGE

#include <sys/socket.h>
#include <netinet/in.h>
s = **socket(AF_INET, SOCK_DGRAM, 0);**

DESCRIPTION

UDP is a simple, unreliable datagram protocol which is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the **sendto** and **recvfrom** calls, though the **connect**(2) call may also be used to fix the destination for future packets (in which case the **recv**(2) or **read**(2) and **send**(2) or **write**(2) system calls may be used).

UDP address formats are identical to those used by TCP. In particular UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (i.e., a UDP port may not be "connected" to a TCP port). In addition broadcast packets may be sent (assuming the underlying network supports this) by using a reserved "broadcast address"; this address is network interface dependent.

DIAGNOSTICS

A **udp** socket operation may fail with one of the following errors returned:

[EISCONN] An attempt was made to establish a connection on a socket which is already connected, or an attempt was made to send a datagram with the destination address of a connected socket specified.

[ENOTCONN] An attempt was made to send a datagram, but no destination address was specified and the socket hasn't been connected.

[ENOBUFS] The system can't allocate enough memory for an internal data structure.

[EADDRINUSE] An attempt was made to create a socket with a port that has already been allocated.

[EADDRNOTAVAIL]
An attempt was made to create a socket with a network address for which no network interface exists.

RELATED INFORMATION

send(2), recv(2), intro(4N), inet(4F)

This is a topical index for Section 4 of the *DOMAIN/IX Programmer's Reference Manual for BSD4.2*. For a permuted index of all reference information, see Appendix A of this manual.

NAME

intro – introduction to file formats

DESCRIPTION

This section describes the formats of various system files that you may need to access, modify, or otherwise understand.

NAME

a.out – cc output

NOTES

The default name for a file produced by the C compiler, cc(1), is *a.out*. The DOMAIN system code generation mechanism produces an *a.out* file that is substantially different from *a.out* files supported on other implementations of the UNIX operating system.

RELATED INFORMATION

ld(1), nm(1)

NAME

aliases – aliases file for **sendmail**

USAGE

/usr/lib/aliases

DESCRIPTION

This file describes user ID aliases used by **/usr/lib/sendmail.** This file is made up of an arbitrary number of lines of the form:

name: name_1, name_2, name_3, . . .

The *name* is the name to alias, and the *name_n* are the aliases for that name. Lines beginning with white space are continuation lines. Lines beginning with # are comments.

Aliasing occurs only on local names. Loops cannot occur, since no message will be sent to any person more than once.

After aliasing has been done, local and valid recipients who have a *.forward* file in their home directory have messages forwarded to the list of users defined in that file.

This is only the raw data file; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the program **newaliases**(1). A **newaliases** command should be executed each time the aliases file is changed for the change to take effect.

NOTES

Because of restrictions in **dbm**(3X), a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by "chaining"; that is, by making the last name in the alias be a dummy name that is a continuation alias.

EXAMPLE

Here's an example of an *aliases* file:

```
##
# Aliases in this file will NOT be expanded in the header from
# Mail, but WILL be visible over networks or from /bin/mail.
#
#       >>>>>>>>>> The program "newaliases" must be run after
#       >> NOTE >> this file is updated or else changes won't
#       >>>>>>>>>> get to sendmail.
##

MAILER-DAEMON:bob
root: bcking

texhax: texhax_list
tusers: t_users_list
msgs: "|/usr/ucb/msgs -s"
sherry: sar
speedo: mr_earl
```

RELATED INFORMATION

DOMAIN/IX Administrator's Reference for BSD4.2 **newaliases(1), dbm(3X), sendmail(8)**

NAME

ar – archive (library) file format

USAGE

#include <ar.h>

DESCRIPTION

The archive command **ar** combines several files into one.

A file produced by **ar** has a magic string at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARMAG   "!<arch>0
#define SARMAG 8

#define ARFMAG "`\n"

struct ar_hdr {
        char        ar_name[16];
        char        ar_date[12];
        char        ar_uid[6];
        char        ar_gid[6];
        char        ar_mode[8];
        char        ar_size[10];
        char        ar_fmag[2];
};
```

The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. The other fields are left-adjusted, blank-padded numbers. They are decimal except for *ar_mode*, which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on a even (0 mod 2) boundary; a newline is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

NOTES

Filenames lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

Archives used mainly as libraries to be searched by the link-editor **ld** have a different format.

RELATED INFORMATION

ar(1), **ld**(1), **nm**(1)

NAME

dir – format of directories

USAGE

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, except 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 inode entry The structure of a directory entry as given in the include file is:

```
#ifndef DEV_BSIZE
#define    DEV_BSIZE  1024
#endif

#define DIRBLKSIZ     DEV_BSIZE
#define    MAXNAMLEN32

struct     direct {
           unsigned long d_ino;
           short    d_reclen;
           short    d_namlen;
           char     d_name[MAXNAMLEN + 1];
};

/*
 * The DIRSIZ macro gives the minimum record length which will hold
 * the directory entry. This requires the amount of space in struct direct
 * without the d_name field, plus enough space for the name with a terminating
 * null byte (dp->d_namlen+1), rounded up to a 4 byte boundary.
 */
#undef DIRSIZ
#define DIRSIZ(dp) \
((sizeof (struct direct) - (MAXNAMLEN+1)) + (((dp)->d_namlen+1 + 3) &~ 3))

/*
 * Definitions for library routines operating on directories.
 */
typedef struct _dirdesc {
           int      dd_fd;
           long     dd_loc;
           long     dd_size;
           char     dd_buf[DIRBLKSIZ];
} DIR;
```

```
#ifndef NULL
#define NULL 0
#endif
extern    DIR *opendir();
extern    struct direct *readdir();
extern    long telldir();
extern    void seekdir();
#define rewinddir(dirp)  seekdir((dirp), (long)0)
extern    void closedir();
```

NOTES

On many UNIX systems, the first two entries in each directory are for . (dot) and .. (dotdot). The first is an entry for the directory itself. The second is for the parent directory. The meaning of dotdot is modified for the root directory of the master file system; there is no parent, so dotdot has the same meaning as dot.

While the dot and dotdot directory entries do not exist in the *bsd4.2* version of DOMAIN/IX, the naming server recognizes . as "this directory" and .. as "the parent directory of this directory." When dot is // (the network root), dot and dotdot are the same.

NAME

fstab – static information about filesystems

USAGE

#include <fstab.h>

DESCRIPTION

The file */etc/fstab* contains descriptive information about the various file systems. On DOMAIN systems, it is a link to *`node_data/etc.fstab*. Programs read this file. They do not write to it. It is created during the installation process.

The order of records in *fstab* is important because **mount**(8) and **umount**(8) sequentially iterate through the file in performing their respective functions.

The special file name is the block special filename, and not the character special filename. If a program needs the character special filename, the program must create it by appending an "r" after the last "/" in the special filename.

If **fs_type** is "rw" or "ro" then the file system whose name is given in the **fs_file** field is normally mounted read-write or read-only on the specified special file.

If **fs_type** is specified as "xx" the entry is ignored. This is useful to show disk partitions that are currently not used.

```
#define FSTAB_RW "rw"    /* read-write device */
#define FSTAB_RO "ro"    /* read-only device */
#define FSTAB_XX "xx"    /* ignore totally */

struct fstab {
        char    *fs_spec;      /* block special device name */
        char    *fs_file;      /* file system path prefix */
        char    *fs_type;      /* rw,ro,or xx */
        int     fs_freq;       /* dump frequency, in days;currently unused */
        int     fs_passno;     /* pass number on parallel dump;currently unused */
};
```

FILES

/etc/fstab	static information on file systems (normally a link to *`node_data/etc.fstab*.

NAME

group – group file

DESCRIPTION

The file */etc/group* contains, for each group, the following information:

- group name
- 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 newline. If the password field is null, no group password is demanded.

This file resides in the */etc* directory, and normally has general read permission.

NOTES

On DOMAIN Systems, the *group* file is created from network registry information by the **crpasswd(8)** program.

FILES

/etc/group group information file

RELATED INFORMATION

setgroups(2), initgroups(3X), passwd(5). crpasswd(8)

NAME

hosts – host name database

DESCRIPTION

The */etc/hosts* file contains information regarding the known DARPA Internet hosts with which your DOMAIN node can communicate (usually via TCP/IP). For each host a single line should be present with the following information:

official host name
Internet address
aliases

Items are separated by any number of blanks and/or tab characters. A # character indicates the beginning of a comment; characters between a # and the next newline are not interpreted by routines which search the file. This file is normally created from the official host database maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional "." notation using the **inet_addr()** routine from the Internet address manipulation library, **inet**(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts database of internet hosts

RELATED INFORMATION

gethostent(3N)

NAME

inetd.conf – configuration file for **inetd**(8C)

DESCRIPTION

This file, nominally */etc/inetd.conf*, is, in nearly all installations, a link to the per-node file *`node_data/etc.inetd.conf*. The Internet superdaemon, **inetd**(8), reads this file at boot time and, in some cases, after it gets a hangup signal.

The *etc.inetd.conf* file is "free format." All fields must be present in each entry, and must appear in the order shown below.

service name	Must be must present in */etc/services*.
socket type	Must be one of stream, dgram, raw, rdm, or seqpacket.
protocol	Must be listed in */etc/protocols*.
wait/nowait	Use *wait* for single-threaded servers (ones that simply take over the socket from **inetd**). Use *nowait* for multi-threaded servers (ones which connect directly to the peer, freeing up the socket for continued use by **inetd**.)
server program	The full pathname to this program (e.g., */etc/ftpd*).
server program arguments	A maximum of MAXARGS (normally 5).

Continuation lines, if required, must begin with a space or tab. To allow comments, **inetd** ignores any line that has a pound sign (#) in column 1.

EXAMPLES

We ship a template for *inetd.conf* with the *bsd4.2* version of DOMAIN/IX. Copy this template from the master DOMAIN/IX node at your site to your node's *`node_data* directory using a command line like the one below.

% **cp** *template_file* **\`node_data/etc.inetd.conf**

where *template_file* is the file */sys/node_data/etc.inetd.conf* on a DOMAIN/IX administrative node at your site. Note that in the C and Bourne shells, you must escape the backquote with a backslash.

The template file includes entries for all internet services available in the *bsd4.2* version of DOMAIN/IX. All entries are commented out in the template file. Unless you remove the comment delimiters, **inetd** will be configured to do nothing. In the example file below, comment lines have been removed from the entries for **telnetd**(8C) and **rlogind**(8C).

```
# etc.inetd.conf template
# DOMAIN/IX version of 12/04/85
#
# remove # characters to allow services
#
#
#  Run telnetd and/or rlogind on nodes to which
#  you wish to allow incoming login
#telnet     stream   tcp   nowait   /etc/telnetd    telnetd
#login      stream   tcp   nowait   /etc/rlogind    rlogind
#
#  Run rshd and/or rexecd on nodes to which
#  you wish to allow remote command execution
#shell      stream   tcp   nowait   /etc/rshd       rshd
#exec       stream   tcp   nowait   /etc/rexecd     rexecd
#
#  Only one ftpd is needed per ring, but you may want to
#  run more than one to maximize availability
#ftp        stream   tcp   nowait   /etc/ftpd       ftpd
```

FILES

/etc/services List of Internet services

/etc/protocols List of Internet protocols

/etc/inetd Internet superdaemon; reads *inetd.conf* for configuration data.

/etc/ftpd FTP daemon

/etc/rexecd Remote execution server

/etc/rlogind Remote log-in daemon

/etc/rshd Remote Shell server

/etc/telnetd DARPA TELNET protocol server

RELATED INFORMATION

inetd(8C), **services**(5), **rexecd**(8C), **rlogind**(8C), **rshd**(8C), **telnetd**(8C),

NAME

mtab – mounted file system table

USAGE

#include <fstab.h>
#include <mtab.h>

DESCRIPTION

On DOMAIN/IX systems, the **mtab** file, */etc/mtab*, is a link to the per-node file *`node_data/etc.mtab*. It is created upon installation of DOMAIN/IX software. The file contains a table of devices mounted by the **mount**(8) command. **Mount** adds entries to this file; **umount**(8) removes them.

The table is a series of **mtab** structures, as defined in *<mtab.h>*. Each entry contains the null-padded name of the place where the special file is mounted, the null-padded name of the special file, and a type field, one of those defined in *<fstab.h>*. The special file has all its directories stripped away; that is, everything through the last slash (/) is discarded. The type field indicates whether the file system is mounted read-only or read-write.

This table is present for reference purposes only. It does not matter to **mount** if there are duplicated entries, nor to **umount** if a name cannot be found.

FILES

/etc/mtab mounted file system table

NOTES

Owners of diskless DOMAIN Nodes can create this file in a *`node_data* directory on their disked partner by running the **mkptnr**(8) command.

RELATED INFORMATION

mount(8)
umount(8)

NAME

networks – network name database

DESCRIPTION

The */etc/networks* file contains information regarding DARPA Internet networks with which your DOMAIN node can communicate. For each host a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A # character indicates the beginning of a comment; characters between a # and the next newline are not interpreted by routines which search the file. This file is normally created from the official host database maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network numbers are specified in the conventional "." notation using the **inet_network()** routine from the Internet address manipulation library, **inet**(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks	database of reachable networks

RELATED INFORMATION

getnetent(3N)

NAME

passwd – password file

DESCRIPTION

Passwd contains, for each user account, the following information:

log-in name
numerical user ID
numerical group ID
full name and uid
initial working directory
program to use as shell

All fields but the last are derived from data in the network registry by the **crpasswd**(1m) program. On DOMAIN Systems, */etc/passwd* exists soleley to provide account information in a form familiar to UNIX programs and users. It is not used in verifying passwords at login time and in fact, it includes no passwords at all.

Each field within a user's entry is separated from the next by a colon. Each user is separated from the next by a newline. Since encrypted passwords are maintained in the registry and not copied into the password file by **crpasswd**, the second field is always null. If the Shell field is null, the Bourne Shell is used.

We supply a program, **crpasswd**(8), that builds */etc/passwd*, */etc/group*, and */etc/passwd.map* from information in the network registry. To add a new user to the system, follow the procedures for creating a new account described the *DOMAIN/IX Administrator's Reference for BSD4.2*, then update the password file by running **crpasswd**. Do not edit the password file unless you need to change the "shell" field. If you do change this field, run **crpasswd** after the change is completed.

EXAMPLE

The line below is a prototypical record in */etc/passwd*.

robinson::uuuu:gg:Sheryl &, xxxxxxxx.xxxxxxxx://home/dir:/bin/csh

This example shows the */etc/passwd* entry for user "Sheryl Robinson." It includes her log-in name, a null field, her user and group ID numbers, her full name and uid (separated by a comma), home directory, and a shell field that specifies the C Shell. (If you include an ampersand in the full name field, it will be expanded into the log-name. This labor-saving feature is, of course, only useful where someone logs in with some portion of their full name.) The uid is a unique numeric identifier derived from the time the account was created and the node ID of the node on which the account was created.

FILES

/etc/passwd	the password file
/etc/passwd.map	uid-to-userid mapping
/etc/group	the group file

RELATED INFORMATION

getpwent(3), login(1), group(5), crpasswd(8)

NAME

phones – remote host phone number database

DESCRIPTION

The file */etc/phones* contains the system-wide private phone numbers for the **tip**(1C) program. Since phone numbers can be privileged information, this file is normally protected against general readability. The format of the file is a series of lines of the form:

system-name phone-number

Where *system-name* is one of those defined in the **remote**(5) file and the *phone-number* is constructed from the set [0123456789-=]. The "=" and "-" characters cause some autodialers to pause.

Only one phone number per line is permitted. However, if more than one line in the file contains the same *system-name*, **tip**(1C) will attempt to dial each one in turn until it establishes a connection.

FILES

/etc/phones phone number database for **tip**(1C)

RELATED INFORMATION

tip(1C), **remote**(5)

NAME

plot – graphics interface

DESCRIPTION

Files of this format are produced by routines described in **plot**(3X), and are interpreted for various devices by commands described in **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. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the "current point" for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in **plot**(3X).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See **plot**(1G).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

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

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are "dotted," "solid," "longdashed," "shortdashed," and "dotdashed." Effective only in **plot 4014** and **plot ver**.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of **plot**(1G). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

RELATED INFORMATION

plot(1G), **plot**(3X), **graph**(1G)

NAME

printcap – printer capability data base

USAGE

/etc/printcap

DESCRIPTION

Printcap is a simplified version of the **termcap**(5) data base. However, **printcap** is used solely to describe line printers. The spooling system reads the **printcap** file every time it is used, allowing you to add and delete printers dynamically. Each entry in the data base is used to describe one printer.

The default printer is normally **lp**, though the environment variable PRINTER may be used to override this. Each spooling utility supports an option, –P*printer,* to allow explicit naming of a destination printer.

Refer to the *DOMAIN/IX Administrator's Reference Manual for BSD4.2* for a more complete discussion of how to set up the database for a given printer.

CAPABILITIES

The layout of this file is identical to the layout of */etc/termcap*.

Name	Type	Default	Description
af	str	NULL	name of accounting file
br	num	none	if lp is a tty, set the baud rate (ioctl call)
cf	str	NULL	cifplot data filter
df	str	NULL	tex data filter (DVI format)
fc	num	0	if lp is a tty, clear flag bits (sgtty.h)
ff	str	\f	string to send for a form feed
fo	bool	false	print a form feed when device is opened
fs	num	0	like 'fc' but set bits
gf	str	NULL	graph data filter (plot (3X) format)
ic	bool	false	driver supports (non standard) ioctl to indent printout (unimplemented)
if	str	NULL	name of text filter which does accounting
lf	str	/dev/console	error logging file name
lo	str	lock	name of lock file
lp	str	/dev/lp	device name to open for output
mc	num	infinite	maximum number of copies allowed
mx	num	1000	maximum file size (in BUFSIZ blocks), zero = unlimited
nd	str	NULL	next directory for list of queues (unimplemented)
nf	str	NULL	ditroff data filter (device independent troff)
of	str	NULL	name of output filtering program

pc	str	NULL	Command to run instead of directing output to **lp** or **rp**. The command should behave like a printer. The value supplied for DOMAIN Systems is: "/com/prf -banner off -text -npag -headers off"
pl	num	66	page length (in lines)
pw	num	132	page width (in characters)
px	num	0	page width in pixels (horizontal)
py	num	0	page length in pixels (vertical)
rf	str	NULL	filter for printing FORTRAN style text files
rm	str	NULL	machine name for remote printer
rp	str	lp	remote printer name argument
rs	bool	false	restrict remote users to those with local accounts
rw	bool	false	open the printer device for reading and writing
sb	bool	false	short banner (one line only)
sc	bool	false	suppress multiple copies
sd	str	/usr/spool/lpd/lp	spool directory
sf	bool	false	suppress form feeds
sh	bool	false	suppress printing of burst page header
st	str	status	status file name
tf	str	NULL	troff data filter (phototypesetter)
tr	str	NULL	trailer string to print when queue empties
vf	str	NULL	raster image filter
xc	num	0	if lp is a tty, clear local mode bits (tty(4))
xs	num	0	like 'xc' but set bits

NOTES

Blank lines in a *printcap* file will cause **lp**-related commands to act as if there is a "nameless" printer defined there.

If the local line printer driver supports indentation, the daemon must understand how to invoke it.

RELATED INFORMATION

lpq(1), **lpr**(1), **lprm**(1), **lpc**(8), **lpd**(8), **termcap**(5), **/com/prf.**

NAME

protocols – protocol name database

DESCRIPTION

The **protocols** file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A # character indicates the beginning of a comment; characters between a # and the next newline are not interpreted by routines that search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/protocols DARPA Internet protocols database

RELATED INFORMATION

getprotoent(3N)

NAME

remote – remote host description file

DESCRIPTION

Information about systems accessible via **tip**(1) and is stored in */etc/remote*, an ASCII file that is structured somewhat like the **termcap**(5) file. Each line in the file provides a description for a single system. Fields are separated by a colon (":"). Lines ending in a \ character with an immediately following newline are continued on the next line.

The first entry is the name(s) of the host system. If there is more than one name for a system, the names are separated by vertical bars. After the name of the system comes the fields of the description. A field name followed by an equal sign ("=") indicates that a string value follows. A field name followed by a pound sign ("#") indicates that a numeric value follows.

Entries named "tip*" and "cu*" are used as default entries by **tip**, and the **cu** interface to **tip**, as follows. When **tip** is invoked with only a phone number, it looks for an entry of the form "tip300", where 300 is the baud rate with at the connection is to be made. When the **cu** interface is used, entries of the form "cu300" are used.

CAPABILITIES

Capabilities described below are either strings (str), numbers (num), or Boolean flags (bool). A string capability is specified by *capability=value*; e.g. "dv=/dev/harris". A numeric capability is specified by *capability#value*; e.g. "xa#99". A Boolean capability is specified by simply listing the capability.

at (str) Auto call unit type. [DOMAIN/IX supports these *values* for **at**: v831 (Racal-Vadic 831), v3451 (Racal-Vadic V3451 or VA212),or ventel (Ventel 212+).

br (num) The baud rate used in establishing a connection to the remote host. This is a decimal number. The default baud rate is 300 baud.

cm (str) An initial connection message to be sent to the remote host. For example, if a host is reached through port selector, this might be set to the appropriate sequence required to switch to the host.

cu (str) Call unit if making a phone call. Default is the same as the **dv** field.

di (str) Disconnect message sent to the host when a disconnect is requested by the user.

du (bool) This host is on a dial-up line.

dv (str) Device(s) to open to establish a connection. If this file refers to a terminal line, **tip**(1) attempts to perform an exclusive open on the device to ensure that only one user at a time has access to the port.

el (str) Characters marking an end-of-line. The default is NULL. Tilde ("~ ") escapes are only recognized by **tip** after one of the characters in **el**, or after a carriage-return.

fs (str) Frame size for transfers. The default frame size is equal to BUFSIZ.

hd (bool) The host uses half-duplex communication, local echo should be performed.

ie (str) Input end-of-file marks. The default is NULL.

oe (str) Output end-of-file string. The default is NULL. When **tip** is transferring a file, this string is sent at end-of-file.

pa (str) The type of parity to use when sending data to the host. This may be one of "even", "odd", "none", "zero" (always set bit 8 to zero), "one" (always set bit 8 to 1). The default is even parity.

pn (str) Telephone number(s) for this host. If the telephone number field contains an @ sign, **tip** searches the file */etc/phones* file for a list of telephone numbers; (See **phones**(5)).

tc (str) Indicates that the list of capabilities is continued in the named description. This is used primarily to share common capability information.

EXAMPLE

This short example demonstrates the use of the capability continuation feature:

```
UNIX-1200:\
        :dv=/dev/sio1:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:br#1200:
arpavax|ax:\
        :pn=7654321%:tc=UNIX-1200
```

FILES

/etc/remote remote dial-up host descriptions

RELATED INFORMATION

tip(1), **phones**(5)

NAME

sccsfile – format of Source Code Control System (SCCS) file

DESCRIPTION

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains log-in names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This is the *control character* and is represented graphically in these pages as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form DDDDD represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

```
@hDDDDD
```

The value of the checksum is the sum of all characters, except those of the first line. The @h provides a *magic number* of (octal) 064001.

Delta table

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
  .
  .
  .
@c <comments> ...
  .
  .
  .
@e
```

The first line (@s) contains the number of lines inserted/deleted/unchanged, respectively. The second line (@d) contains the type of the delta (currently, normal: D, and removed: R), the SCCS ID of the delta, the date and time the delta was created, the log-in name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the @c lines contain comments associated with the delta.

The @e line ends the delta table entry.

User names

The list of log-in names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these log-in names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta. Any line starting with a ! prohibits the succeeding group or user from making deltas.

Flags Keywords used internally (see **admin**(1) for more information on their use). Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t <type of program>
@f v <program name>
@f i <keyword string>
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default SID>
@f n
@f j
@f l <lock-releases>
@f q <user defined>
@f z <reserved for use in interfaces>

The **t** flag defines the replacement for the %Y% identification keyword. The **v** flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number validity checking program. The **i** flag controls the warning/error aspect of the "No ID keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **-b** keyletter may be used on the **get** command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the %M% identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a **get** command. The **n** flag causes **delta** to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes **get** to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing (get(1) with the **-e** keyletter). The **q** flag defines the replacement for the %Q% identification keyword. The **z** flag is used in certain specialized interface programs.

Comments

Arbitrary text is surrounded by the bracketing lines @t and @T. The comments section typically will contain a description of the file's purpose.

Body The body consists of text lines and control lines. Text lines do not begin with the control character, control lines do. There are three kinds of control lines: *insert*,*delete*, and *end*, represented by:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

respectively. The digit string is the serial number corresponding to the delta for the control line.

RELATED INFORMATION

admin(1), delta(1), get(1), prs(1)

NAME

services – database of Internet services

DESCRIPTION

The */etc/services* file contains information regarding the known services available in the Internet. Each service description consists of a single line that includes the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*. A / separates the port and protocol (e.g. "512/tcp"). A # indicates the beginning of a comment; characters between a # and the next newline are not interpreted by routines that search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/services database of Internet services

RELATED INFORMATION

getservent(3N)

NAME

tar – tape archive file format

DESCRIPTION

Tar(1) (the tape archiver command) dumps several files into one, typically on a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block that describes the file, followed by zero or more blocks that give the contents of the file. At the end of the tape, two blocks filled with binary zeros serve as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the **tar** command line — default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads.

The header block looks like:

```
#define TBLOCK     512
#define NAMSIZ     100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

Name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width *w*) contains *w*-2 digits, a space, and a null, except for *size* and *mtime*, which do not contain the trailing null. *Name* is the name of the file, as specified on the **tar** command line. Files dumped because they were in a directory that was named in the command line have the directory name as prefix and */filename* as suffix. *Mode* is the file mode, with the high bit masked off. *Uid* and *gid* are the user and group numbers which own the file. *Size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *Mtime* is the modification time of the file at the time it was dumped. *Chksum* is a decimal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *Linkflag* is ASCII zero if the file is "normal" or a special file, ASCII 1 if it is an hard link, and ASCII 2 if it is a symbolic link. The name linked to, if any, is in *linkname,* with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be manually re-scanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

NOTES

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

RELATED INFORMATION

tar(1)

NAME

termcap – terminal capability database

USAGE

/etc/termcap

DESCRIPTION

Termcap is a database describing terminals, used, e.g., by **vi**(1) and **curses**(3X). This file includes definitions of the capabilities of various terminals, and details about how these terminals handle various operations. Padding requirements and initialization sequences are included in **termcap.**

Entries in **termcap** consist of a number of colon-separated fields. The first entry for each terminal gives the names known for the terminal, separated by | characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a system-wide database. 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 that 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 zero to last column
CC	str		Command character in prototype if terminal settable
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays same
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only
da	bool		Display may be retained above

dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed
dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give ":ei=:" if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give ":im=:" if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by "other" function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of "keypad transmit" mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of "other" keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in "keypad transmit" mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on "other" function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor
ms	bool		Safe to move while in standout and underline mode
mu	str		Memory unlock (turn off memory lock).

nc	bool		No correctly working carriage return (DM2500, H2000)
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default \n)
ns	bool		Terminal is a CRT but doesn't scroll
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with **is**)
rc	str		Restore cursor position, type, and attributes
sc	str		Save cursor position, type, and attributes
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)
ta	str	(P)	Tab (other than ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't overstrike
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)

NAME

types – primitive system data types

USAGE

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in the system code; some data of these types are accessible to user code:

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by **uuencode**(1) consist of a header line, followed by a number of body lines, and a trailer line. **Uudecode**(1) will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The first 6 characters of the header line must be the string "begin". This string is followed by a mode (in octal) and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

RELATED INFORMATION

uuencode(1), uudecode(1), uusend(1), uucp(1), mail(1)

This is a topical index for Section 5 of the *DOMAIN/IX Programmer's Reference Manual for BSD4.2*. For a permuted index of all reference information, see Appendix A of this manual.

Appendix A: Permuted Index

This permuted index covers reference material in the *DOMAIN/IX Command Reference Manual*, the *DOMAIN/IX Programmer's Reference Manual*, and parts of *System Administration for DOMAIN/IX*. In addition, there is a topical index located at the end of each section of these manuals.

READER'S RESPONSE FORM

We use readers' comments in revising and improving our documents.

DOMAIN/IX Programmer's Reference Manual for BSD4.2, Order Number 5801, Revision 01
Date of Publication: December 1986

What is the best feature of this manual?

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible.)

What type of user are you?

___ Systems programmer; language

___ Applications programmer; language

___ Manager/Professional

___ Administrative/Support Personnel

___ Student programmer

___ User with little programming experience

___ Other

How often do you use the DOMAIN system?

Nature of your work on the DOMAIN system:

Your name Date

Organization

Street Address

City State Zip/Country

No postage necessary if mailed in the U.S. Fold on dotted lines (see reverse side), tape, and mail.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD