## NAME

ldiv – long division

## SYNOPSIS

**ldiv (hidividend, lodividend, divisor)**

**lrem (hidividend, lodividend, divisor)**

## DESCRIPTION

These routines are provided for compatibillity with existing programs only; the **long** C variable type should be used instead.

The concatenation of the signed 16-bit *hidividend* and the unsigned 16-bit *lodividend* is divided by *divisor*. The 16-bit signed quotient is returned by *ldiv* and the 16-bit signed remainder is returned by *lrem*. Divide check and erroneous results will occur unless the magnitude of the divisor is greater than that of the high-order dividend.

An integer division of an unsigned dividend by a signed divisor may be accomplished by:

**quo = ldiv(0, dividend, divisor);**

and similarly for the remainder operation.

Often both the quotient and the remainder are wanted. Therefore *ldiv* leaves a remainder in the external cell *ldivr*.

## NOTE

These routines are obsolete and should not be used. Use **long** variable types instead.

## BUGS

To reiterate, *ldiv* will not work if the magnitude of the divisor is less than the high order dividend. There is no check for this condition.

## NAME

lfs − Logical File System operations

## SYNOPSIS

```
#include <lfsh.h>
lfmount (name, bname, flag)
lfumount (name)
lfopen (name, mode)
lfclose ( )
lfcreate (fd, lfn, nsectors)
lfdelete (fd, lfn, nsectors)
lfread (fd, lfn, sector, buf, size)
lfwrite (fd, lfn, sector, buf, size)
lfswitch (fd, lfn, lfn2)
lfsize (fd, lfn)
lffde(fd, lfn, buf)
lfufde(fd, lfn, buf)
lfstat ( )
lfistat ( )
lfupdate( )

unsigned short lfn, lfn2;
char *name, *bname, *buf;
```

## DESCRIPTION

The Logical File System is a pseudo device which provides a fast-access file system implemented through the UNIX raw I/O facility. It uses the *ioctl*(2), *open*(2), and *close*(2) system calls to interface to the user process level.

The file system provided by *lfs* consists of a set of contiguously allocated files, specified by their file number (lfn), which are manipulated by the Logical File System operations. The lfn is used as an index into a file definition table stored on disk whose entries record the physical location and size of each file. File definition entries are read into memory when referenced and remain there if frequently used. The 512 byte block (i.e. one disk sector) is the basic unit of size for operations with the Logical File System.

The calling parameter definitions are:

*name* The name of the character-special file by which the LFS driver is accessed.

*bname* The name of the block-special file for the LFS disk area desired.

*flag* If non-zero, indicates that the file system is to be mounted read-only.

*mode* The mode to be used when the Logical File System is opened (see *open*(2)); the user must have the requested access permissions with respect to the UNIX file *name*.

*lfn* The file number of the file within the logical file system to be manipulated. Up to 65,535 files are permitted per file system.

*nsectors* The number of 512 byte blocks.

*sector* Position within a logical file in terms of 512 byte disk blocks.

*buf* Buffer address for reading or writing data.

*size* Buffer size in 512 byte blocks.

The *lfs* operations are:

**lfmount** Associates character device *name* used to access the LFS driver with the block device *bname* corresponding to the LFS disk area. This operation can be thought of as

"mounting" *bname* on *name*. If *flag* is non-zero, *bname* is mounted read-only. If either name is already in the internal mount table, or if the LFS header for *bname* cannot be accessed or contains the wrong magic number, a −1 is returned.

**lfumount** Disassociates *name* with its current block device. If the device is not in the mount table, or is currently open, a −1 is returned.

**lfopen** Opens the Logical File System associated with *name*. Normally, a file descriptor is returned; however, if *name* does not exist or does not appear in the mount table, a −1 is returned.

**lfclose** Closes the passed file descriptor previously returned by *lfopen*. All modified file definition entries for the specified device are flushed to disk.

**lfcreate** Creates logical file *lfn* in the LFS specified by *fd*. *Nsectors* is the initial size in sectors of the file. If *nsectors* is **0**, a file **0** sectors long is created (whereas *lfdelete* deletes the file given the same input). All storage is allocated to *lfn* and remains so (even if never written to) until *lfcreate* or *lfdelete* is called to change the file size. Legal *lfn*'s start with **1**; if *lfn* is **0** in an *lfcreate* call, the number of a 'scratch' logical file is returned. By convention, scratch files are allocated starting from the end of the range of legal *lfn*'s and proceed downward. *Lfcreate* may be used to decrease or increase the size of an existing file. Because all *lfn*'s are contiguous, increasing the size of an *lfn* may force a copy of the entire file. Increasing or decreasing file sizes may also lead to fragmentation of the storage freelist. Normally, *Lfcreate* returns the *lfn* of the file created; a −1 returned indicates an error.

**lfdelete** Changes the size of logical file *lfn* in the LFS specified by *fd* to be *nsectors*. If *nsectors* is **0** the file is deleted. Normally, the *lfn* of the file deleted is returned; a −1 returned indicates an error.

**lfread** *Size* sectors are read from logical file *lfn* in the LFS specified by *fd* and placed into *buf*. *Sector* is the starting position within *lfn* for the read. Normally, the number of sectors read is returned; this can be less than *size* if the end of the file is encountered. A −1 returned indicates an error.

**lfwrite** *Size* sectors are written from *buf* to logical file *lfn* in the LFS specified by *fd*. *Sector* is the starting position within *lfn* for the write. Normally the number of blocks written is returned; this can be less than *size* if the end of the file is encountered. A −1 returned indicates an error, e.g., if *sector* is beyond the end of the file.

**lfswitch** Switches the physical storage allocated for logical files *lfn* and *lfn2* in the LFS specified by *fd* without copying. This enables files to be replaced on-line very quickly. A −1 returned indicates an error.

**lfsize** Returns the size in sectors of *lfn*. A −1 returned indicates an error.

**lffde** Retrieves the file definition entry for logical file *lfn* in the LFS specified by *fd* . This includes flag bits, the file starting location and size, and user information stored therein.

**lfufde** Stores the user portion of the file definition entry for logical file *lfn* in the LFS specified by *fd*. This information is arbitrary and may be changed by the user.

**lfstat** Prints several statistics about LFS operations. These counts are summed over *all* mounted LFS. The statistics printed are: the number of *lfs* calls, the number of raw read and write calls, the number of raw read and write blocks transferred, the number of file definition entries found in memory, the number of such entries retrieved from disk, the hit ratio for retrieving such entries, and the number of calls to the UNIX block I/O system for administering the raw disk area.

**lfistat** Zeroes all *lfs* statistics counts.

**SEE ALSO**

mklfs(1M), close(2), ioctl(2), open(2), lfs(5)

*The Logical File System — A Fast-Access File System Using UNIX Raw I/O*, TM 79-9471-1, J. R. McSkimin.

**AUTHOR**

J. C. Kaufeld Jr.
J. R. McSkimin

## NAME

lib7 — Version 7 library

## SYNOPSIS

**occ ... -l7**

## DESCRIPTION

This library provides conversion routines which allow a C program written assuming a version 7 file system environment to execute in an actual version 6 environment. It is designed for use in conjunction with the old C compiler and libraries in adapting a recent program for executing on an older operating system.

The conversions provided give the system interface indicated in the **:o** pages of the manual, while maintaining the user interface indicated in the regular manual pages. A list of the conversions included follows:

**chgrp**
**chown**
**fstat**
**getgid**
**getuid**
**gtty**
**ioctl**
**isatty**
**longjmp**
**lseek**
**stty**
**setjmp**
**stat**
**ttyname**
**utime**

## FILES

/lib/lib7.a

## SEE ALSO

intro(3C), libi1(3X).

## NAME

libi1 − CB UNIX Release 1 Conversion Library

## SYNOPSIS

**occ** ... **-l7 -li1**

## DESCRIPTION

This library provides conversion routines which allow a C program written assuming a version 7 file system environment to execute in an environment identical to that provided in CB UNIX Release 1.0. It is designed for use in conjunction with *lib7.a* together with the old C compiler and libraries in adapting a recent program for executing on the previous release, perhaps in field systems. In particular, this library disables some newly added features which were not available on the Release 1.0 system, so that possible incompatibilities can be detected at link time.

The following conversions are made:

**execle**
**execve** Converted to **execl** and **execv** with no environment variables.

**fcntl** Converted to use **dup** for file descriptor duplication; emulate autoclose where possible.

**chroot**
**getesw**
**getu** CB UNIX Release 1 system call interface provided (differs from that in CB UNIX Release 2).

**reboot**
**sprofil**
**lock**
**noulk**
**rdsem**
**setsem**
**unlock** Functions disabled (unavailable in CB UNIX Release 1).

**umask** Dummy routine supplied.

## FILES

/lib/libi1.a

## SEE ALSO

intro(3C), lib7(3X).

## DIAGNOSTICS

Messages of the form "undefined: _noreboo" are generated from the loader if an unimplemented subroutine is used.

## NAME

lnxx − return name of current terminal

## SYNOPSIS

**lnxx (filedes)**

## DESCRIPTION

*Lnxx* searches for the last two characters (if 2 char field option elected) or the last character (if 1 char field selected) of the terminal's name which is specified by the argument *filedes* (file descriptor of a file). If *kk* is returned, the terminal name is then "**/dev/ln***kk*". If *k* is returned, the terminal name is then "**/dev/ln***k*".

**xx** is returned if the indicated file does not correspond to a terminal.

## NOTE

This routine has been replaced in newer versions of the library by *ttyname*(3C).

## NAME

locv — long output conversion

## SYNOPSIS

**char *locv (hi, lo)**
**int hi, lo;**

## DESCRIPTION

*Locv* converts a signed double-precision integer, whose parts are passed as arguments, to the equivalent ASCII character string and returns a pointer to that string.

## NOTE

This routine has been dropped from newer versions of the library. Use scanf(3S) instead.

## NAME

lpdata − decode line printer data files (printers and qmap)

## SYNOPSIS

```
lpdata (name,group,type,buffer,fp)
char *name;          /* printer or queue name */
int group;           /* group associated with printer or queue */
char type;           /* 'p' for the printers file, or 'q' for qmap file */
union
{
 struct printers *buffer;
 struct qmap *buffer;
} ;                  /* buffer for decoded data */
FILE *fp;            /* file pointer to qmap or printers */
```

## DESCRIPTION

*Lpdata* retrieves the information from the *printers* and *qmap* files, and places it in a structured format as given by *lpss.h*. If *name* is not zero, *lpdata* seeks to the beginning of the file *fp* and searches for the specified *name*. The search will be successful only in the cases where the entry has been assigned to *group* or to ANYGID or has been defaulted. If *name* is zero, the next entry assigned to *group* is returned. Upon exhaustion of assigned entries, *lpdata* begins returning entries for the default group. It is necessary to seek to the beginning of *fp* in order to cause *lpdata* to reinitiate its search path. The space for *buffer* should be reserved in the calling program by declaring *buffer* to be either struct printer (in the case of the printers file) or struct qmap (in the case of the qmap file).

## FILES

/usr/lpd/.qmap
/usr/lpd/.printers

## SEE ALSO

lpr(1)

## DIAGNOSTICS

*lpdata* returns −1 if the queue in question cannot be found, or the end of file is reached. In cases of syntax errors in the file, −2 is returned.

## NAME

lpropen – open pipe to the line printer

## SYNOPSIS

```
FILE *lpropen(lprstr,ofp)
char *lprstr;
FILE *ofp;
```

## DESCRIPTION

*Lpropen* returns a file pointer used to write to the line printer queue specified by *lprstr*. If *lprstr* is 0, a default queue is used. *Ofp* is the file pointer used to report any errors encountered as well as the spooling system JOB ID. If *ofp* is 0, all messages are discarded.

Since *lpropen* spins off the lpr program, to insure that the output from the lpr program is not intermixed with other output, it is recommended that the file pointer returned be explicitly closed, and that a wait be executed when all output for the printer has been generated.

## SEE ALSO

lpr(1)

## DIAGNOSTICS

0 is returned if the specified queue does not exist, a pipe cannot be created, or a fork cannot be executed.

## BUGS

Though *lpropen* only returns one file pointer, two descriptors are required for a short time in order ro set up a pipe to the spooling system.

**NAME**

ltod − double precision integer to floating point conversion

**SYNOPSIS**

**double ltod(t)**
**int t[2];**

**DESCRIPTION**

*Ltod* converts a signed double precision (i.e. **long**) integer to the equivalent floating point number.

**LIBRARY**

/lib/libc.a

## NAME

ltoi – long integer to integer conversion

## SYNOPSIS

```
ltoi (lng)
long lng;
```

## DESCRIPTION

*Ltoi* converts a long integer to a single precision integer by truncating the high order word.

## NOTE

This routine has been dropped from later libraries; use type casting instead:

```
long a;
...
... (int) a ...
```

## NAME

malloc, free, realloc, calloc − main memory allocator

## SYNOPSIS

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

**free (ptr)**
**char *ptr;**

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

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

## DESCRIPTION

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

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

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

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

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

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

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

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

## SEE ALSO

break(2)

## DIAGNOSTICS

*Malloc, realloc* and *calloc* return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When *realloc* returns 0, the block pointed to by *ptr* may be destroyed.

## NAME

malloc, free − core memory allocator

## SYNOPSIS

**char *malloc(size)**

**free(ptr)**
**int *ptr;**

## DESCRIPTION

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

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

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

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

## DIAGNOSTICS

*Malloc* returns a NULL (0) if there is no available memory.
Exit with the message "corrupt arena" means you have stored outside the bounds of a block. To get a core dump, use *adb*(1) to plant a breakpoint on *exit*(2).

## NAME

mkdir − make directory

## SYNOPSIS

**mkdir(file,owner,mode,group)**
**char *file;**
**int owner, mode, group;**

## DESCRIPTION

*Mkdir* will make a directory, link the necessary . and .. pointers and set the specified mode, owner, and group based on the following arguments:

*file* A pointer to a string representing a full or partial pathname of a directory to be made.

*owner* An integer representing the owner of the made directory.

*mode* An integer representing the mode of the directory. The mode represents a value acceptable to a chmod system call.

*group* An integer representing the group id of the made directory.

**Mkdir** returns:

**0** directory successfully made.

**-1** file already exists.

**-2** Cannot do a mknod, link, chown, chmod or chgrp or the effective uid is not super user.

The subroutines *chgrp*(2), *chmod*(2), *chown*(2), *getuid*(2), *link*(2), *mknod*(2) and *stat*(2) are used by *mkdir*.

If mkdir returns with a −2, then any work it has done is still there, e.g. if it cannot do a chown, the directory that the mknod and linking has created prior to the chown still exists.

## SEE ALSO

rmdir(3C)

## DIAGNOSTICS

A return code of −2 is serious because it means that *mknod* has done some but not all of its work.

## BUGS

*Mkdir* should not require the effective user id to be super user.

If the requested action cannot be performed, *mkdir* should undo whatever has been done.

NAME

mktemp — make a unique file name

SYNOPSIS

**char *mktemp (template)**
**char *template;**

DESCRIPTION

*Mktemp* replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with six trailing Xs, which will be replaced with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2)

BUGS

It is possible to run out of letters.

## NAME

mktemp − make temporary file name

## SYNOPSIS

**char *mktemp (str)**
**char *str;**

## DESCRIPTION

*Mktemp* creates a unique temporary file name from its argument. The string must contain a substring of trailing capital X's. These X's are replaced by the users process id. A *stat*(2) is then performed to see if the file exists. If it does, another name is generated, and the process is repeated. (Normally this should only be necessary if the system's process ids have wrapped around). In any case, *mktemp* returns a string containing the temporary file name.

## SEE ALSO

mktmp(3)

## DIAGNOSTICS

*Mktemp* returns the string / if it is unable to find a usable

## NAME

mktmp − make a temporary file

## SYNOPSIS

**mktmp** ( )

## DESCRIPTION

*Mktmp* returns a file descriptor of an unnamed file which may be used as a read-write scratch file. When closed all traces of the file disappear.

## FILES

**tmp/**<*pid*> <*pid*> is the process id of the calling process.

## NOTE

*Mktmp* has been replaced in newer libraries by *tempfile*(3S).

## DIAGNOSTICS

A −1 is returned if the file cannot be created.

## NAME

monitor − prepare execution profile

## SYNOPSIS

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

## DESCRIPTION

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

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

To profile the entire program, it is sufficient to use

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

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

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

```
monitor(0);
```

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

## FILES

mon.out

## SEE ALSO

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

## NAME

msg, msgenab, msgdisab, send, sendw, recv, recvw, msgstat, msgctl − old message veneer for sending and receiving messages.

## SYNOPSIS

```
# include <sys/ipcomm.h>
msgenab ( )
msgdisab ( )

send (buf, size, topid, type)
sendw (buf, size, topid, type)
char *buf;

recv (buf, size, &mstruct, type)
recvw (buf, size, &mstruct, type)
char *buf;
struct mstruct mstruct;

msgstat (&mstat, sizeof (mstat), pid)
struct mstat mstat;

msgctl (pid, command, arg)
```

## DESCRIPTION

The routines described here implement the old message interface. They are now implemented as a veneer using the new message implementation. (See *message*(2)).

A process that has enabled message reception has a message queue on which are placed messages sent to it by other processes. Messages are placed on the queue in the order of arrival. The process actually receives a message by requesting a one from the queue. A process may send a message to any other process that has enabled message reception, as long as the receiver does not have an excessive number of messages pending on its queue.

**msgenab** ()
Enable message reception by creating a message queue with the name equal to the *process id*.

**msgdisab** ()
Disable message reception. Any messages still on the queue are destroyed or returned to sender depending upon the *type*.

**send (buf, size, topid, type)**
Send the message in *buf* and of *size* bytes to the process whose process id is *topid*. The message is stored with the specified *type*, which ranges from *1* to *128*. If the queue for the receiving process is full or if there is no more message space in the operating system, return immediately to the sending process with an appropriate error. When *send* is successful, it returns the number of bytes actually sent in the message.

**sendw (buf, size, topid, type)**
Send a message in the same fashion as *send*, but if there isn't room for the message, suspend execution until the message can be sent. When *sendw* is successful, it returns the number of bytes actually sent in the message.

**recv (buf, size, &mstruct, type)**
Receive the first message on the queue if *type* is 0, otherwise receive the first message on the queue whose type matches *type*. Store the message in *buf*, truncating it if the message is larger than *size*. *mstruct* will be filled with the queue name that the sending process has enabled and the actual type of the message. If the sending process did not have messages enabled the queue name in the *mstruct* structure will be 0. If there is no message of the specified type, return immediately with an appropriate error message. Upon a successful

*recv*, the size of the message received is returned.

**recvw (buf, size, &mstruct, type)**
Receive a message in the same fashion as *recv*, but if there isn't a message satisfying the requested *type*, suspend execution until one arrives. Upon a successful *recvw*, the size of the message received is returned.

**msgstat (&mstat, sizeof(mstat), pid)**
Retreive the number of messages currently present on the queue *pid* and the maximum allowable number of messages for this queue. Put the results in the structure *mstat*.

**msgctl (pid, command, arg)**
Perform the specified command on queue *pid*. The only command currently available is **SETMQLEN**, which allows the maximum number of messages that may queue up for a specific process to be adjusted to *arg*.

The number of bytes actually sent or received is returned by *send*, *sendw*, *recv*, and *recvw*.

The *type* argument is used by a sender to assign a type number (1 to 128) to a message. By convention, types 1 to 63 imply that an acknowledgement message is desired; types 64 to 128 imply no acknowledgement is necessary; type 128 is an acknowledgement message. If a process disables messages (or exits) with any messages still on its queue, those of type 1 to 63 are changed to type 128 and, if possible, returned to the sender; those of type 64 to 128 are discarded.

**ipcomm.h** is included here for convenience.

```
/*              @(#)ipcomm.h  3.3               */

/*
 * Interprocess Communication Control Structures
 */

#ifdef KERNEL
/*
 * common flags
 */

#define IP_PERM         03                              /* scope permission mask */
#define IP_ANY 0                                /* system scope */
#define IP_UID  01                              /* userid scope */
#define IP_GID  02                              /* groupid scope */
#define IP_QWANT        0100                    /* entry in msg queue wanted */
#define IP_WANTED       0200                    /* resource is desired */

struct ipaword
{               char            ip_flag;
                char            ip_id; };

/*
 * message control
 */

#define PMSG    5                               /* message sleep priority */
#define MSGIO  02                               /* tell iomove() this is msg */
#define MSGIN  0                                /* same as B_WRITE */
#define MSGOUT          01                              /* same as B_READ */

#define MDISAB          0
#define MENAB 1
#define MSEND 2
```

```
#define MSENDW          3
#define MRECV 4
#define MRECVW          5
#define MSTAT 6
#define MSGCTL          7

struct msghdr
{       struct msghdr   *mq_forw;
        int             mq_size;
        int             mq_sender;
        int             mq_type;
};
struct msgqhdr
{       struct msghdr   *mq_forw;       /* note same position as in msghdr */
        struct msghdr   *mq_last;
        int             *mq_procp;
        char            mq_flag;
        char            mq_cnt;
        int             mq_meslim;
};

#endif

/* commands for msgctl call here */
#define SETMQLEN 0                                      /*set mes q length command*/


struct mstat {
        unsigned        ms_cnt;
        unsigned        ms_maxm;
};

struct mstruct {
        int             ms_frompid;
        int             ms_type;
};
```

## DIAGNOSTICS

An error occurs when enabling messages if no queue is available for use; it is also erroneous to attempt to disable message reception if it is not enabled. When trying to send messages, errors occur because the message is too long, the receiver has not enabled message reception, the type specified is not valid, the receiver has an excessive number of messages outstanding on its queue, or, for *send*, the system message buffers are temporarily full. When receiving messages, errors may occur because the process has not enabled message reception, the requested type or size are invalid, or, for *recv*, a message of the requested type is not on the queue. It is also illegal to set the message limit (via *msgctl*) to a value larger than defined by MAXMSGDEF in **param.h**.

## FILES

/usr/include/sys/ipcomm.h

## BUGS

There is one noticable difference between this veneer and the real old messages. The process id of the sender was always given to the message receiving process even if the sender didn't have messages enabled. Now, if the sender doesn't have messages enabled, the receiver gets a 0.

## SEE ALSO

message(2)

## NAME

nargs — argument count

## SYNOPSIS

**nargs ( )**

## DESCRIPTION

*Nargs* returns the number of actual parameters supplied by the caller of the routine which calls *nargs*.

The argument count is accurate only when none of the actual parameters is *float* or *double*. Such parameters count as four arguments instead of one. Similarly, array arguments will be misinterpreted.

## NOTE

*Nargs* has been dropped from newer libraries due to its limitations. There is no currently available replacement.

## BUGS

This routine does not work at all for programs which run with separated I&D space.

## NAME

nlist − get entries from name list

## SYNOPSIS

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

## DESCRIPTION

*Nlist* examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types, locations and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type, location and value of the name are inserted in the next three fields. If the name is not found, all three entries are set to 0. See *a.out*(5) for a discussion of the symbol table structure.

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

## SEE ALSO

a.out(5)

## DIAGNOSTICS

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

## NAME

nlist − get entries from name list

## SYNOPSIS

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

## DESCRIPTION

*Nlist* examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of a list of 8-character names (null padded) each followed by two bytes and a word. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type, location and value of the name are placed in the two bytes and word following the name. If the name is not found, the type and location entries are set to −1.

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

## SEE ALSO

a.out(5)

## DIAGNOSTICS

All type and location entries are set to −1 if the file cannot be found or if it is not a valid namelist. In addition, a −1 is returned.

## NAME

perror, sys_errlist, sys_nerr, errno — system error messages

## SYNOPSIS

```
perror (s)
char *s;

int sys_nerr;
char *sys_errlist[ ];

int errno;
```

## DESCRIPTION

*Perror* produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno,* which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

## SEE ALSO

intro(2)

## NAME

popen, pclose − initiate I/O to/from a process

## SYNOPSIS

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

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

## DESCRIPTION

The arguments to *popen* are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

## SEE ALSO

pipe(2), fopen(3S), fclose(3S), system(3S), wait(2)

## DIAGNOSTICS

*Popen* returns a null pointer if files or processes cannot be created, or the Shell cannot be accessed.

*Pclose* returns −1 if *stream* is not associated with a 'popened' command.

## BUGS

Only one stream opened by *popen* can be in use at once.

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush*, see *fclose*(3S).

## NAME

printf, fprintf, sprintf − formatted output conversion

## SYNOPSIS

```
#include <stdio.h>

printf (format [, arg ] ... )
char *format;

fprintf (stream, format [, arg ] ... )
FILE *stream;
char *format;

sprintf (s, format [, arg ] ... )
char *s, format;
```

## DESCRIPTION

*Printf* places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places 'output' in the string *s*, followed by the character **\0**. The string *s* must be long enough.

Each of these functions converts, formats, and prints each *arg* under control of the *format*. The *format* is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg*.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- an optional minus sign − which specifies *left adjustment* of the converted value in the indicated field;
- an optional zero which specifies that zero-padding will be done instead of blank-padding;
- an optional digit string specifying a *field width;* if the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment indicator has been given) to make up 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* which gives the number of digits to appear after the decimal point, for e- and f-conversion; the maximum number of significant figures, for g-conversion; or the maximum number of characters to be printed from a string; it also serves as a modifier in o- and x-conversion;
- an optional **l** or **h**, specifying that a following **d**, **i**, **o**, **x**, or **u** corresponds to a long integer (for **l**) or a short integer (for **h**) *arg*.
- a character which indicates the type of conversion to be applied.

A field width or precision may be * instead of a digit string. In this case an integer *arg* supplies the field width or precision. If the integer corresponding to a precision has the value −1, the effect is as if the precision and its preceding decimal point were both absent.

If the end of the *format* occurs between a % and its following format code, that entire format item is ignored.

The conversion characters and their meanings are:

**d** **i** **o** **x** **X** The integer *arg* is converted to decimal (for either **d** or **i**), **octal**, or hexadecimal notation respectively. The letters **abcdef** are used for **x**- conversion, and the letters **ABCDEF** for **X**- conversion. If the *precision* is present, a single leading zero will be prepended to a non-zero value in o-conversion, and the string '0x' (or '0X') will be prepended to the value in x- (X-) conversion.

**f** The float or double *arg* is converted to decimal notation in the style '[−]ddd.ddd' where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed, unless left-justification and zero-padding are both specified, and the field width is strictly larger than the minimum required.

**e** **E** The float or double *arg* is converted in the style '[−]d.ddde±dd' where there is one digit before the decimal point and the number of digits after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. If left-justification and zero-padding are both specified, any zeroes so generated will appear before the **e** (or **E**). If the precision is zero and no padding zeroes are generated on the right, no decimal point will appear.

**g** **G** The float or double *arg* is printed in style **d**, in style **f**, or in style **e**(or **E** in the case of a **G** format code), whichever gives the requested precision in minimum space.

**c** The character *arg* is printed if it is not **\0**.

**s** *Arg* is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is missing all characters up to a null are printed.

**u** The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 to 65535 for integer values, or 0 to 4294967296 for long values).

**%** Print a %; no argument is converted.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by calling *putchar*(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 $\pi$ to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

## SEE ALSO

ecvt(3C), putchar(3S), scanf(3S), stdio(3S).

## NOTES

For compatibility with earlier versions of *printf*, the format codes **D**, **O**, and **U** are currently implemented to mean the same as **ld**, **lo**, and **lu**. These usages should be avoided.

## BUGS

Outrageous precision specifications on **e**, **f**, and **g** formats can cause failure.

## NAME

printf – formatted print

## SYNOPSIS

**printf(fmt, arg$_1$, ...);**
**char *fmt;**

## DESCRIPTION

*Printf* converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf*.

Each conversion specification is introduced by the character %. Following the %, there may be

- an optional minus sign "–" which specifies *left adjustment* of the converted argument in the indicated field;
- an optional digit string specifying a *field width*; if the converted argument has fewer characters than the field width it will be padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the digit string is preceded with the character "0", the padding character will be the character "0". In this case the number is not interpreted as octal. If the digit string is not preceded with a zero, the padding character is the default character which is blank unless the "%>" option has previously been used.
- an optional period "." which serves to separate the field width from the next digit string;
- an optional digit string *(precision)* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are

**d**

**o**

**x** The integer argument is converted to decimal, octal, or hexadecimal notation respectively.

**D**

**O**

**X** The long integer argument is converted to decimal, octal, or hexadecimal notation respectively.

**f** The argument is converted to decimal notation in the style "[–]ddd.ddd", where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed. The argument should be *float* or *double*.

**e** The argument is converted in the style "[–]d.ddde±dd", where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The argument should be a *float* or *double* quantity.

**c** The argument character or character-pair is printed if non-null.

**s** The argument is taken to be a string (character pointer), and characters from the string are printed until a null character or until the number of characters indicated by the

precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed. If the string pointer itself is null, then the string "**(null)**" will be printed.

**u**

l The argument is taken to be an unsigned integer which is converted to decimal and printed (the result will be in the range 0 to 65535).

r The argument is taken to be the base address of a vector which contains a remote argument list. The first element in the list is a character string which replaces the current format. The remaining elements in the vector are accessed and converted as specified by the new format. A reversion to the original format will never occur; thus any characters in the original format following the "**%r**" will be ignored.

> The next character in the string "*fmt*" will replace the default padding character for the remainder of the string unless changed once more through the use of "**>**".

* The next argument is taken to be a field width specification and is used accordingly. For example, "**%*d**" with *x* and *y* as the corresponding arguments in the argument list would be interpreted as specifying as decimal number *y* to be padded to a field width of *x*.

If no recognizable character appears after the %, that character is printed; thus % may be printed by use of the string %%. 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.

*Printf* is actually an interface to the C library *format* subroutine which performs all the necessary formatting. The *format* subroutine is called identically to printf with the exception of an additional argument preceding *printf*'s *fmt* argument. This new argument is the address of the subroutine to be called for every character of output generated by *format*.

If *printf* were written as a C subroutine it would thus appear as follows:

```
printf(fmt)
char *fmt;
{
extern putchar();

        return(format(putchar, "%r", &fmt));
}
```

## SEE ALSO

putchar(3)

## BUGS

Very wide fields (>128 characters) fail.

*Format*, (and consequently *printf*), is not recursive.

## NAME

putc, putchar, fputc, putw − put character or word on a stream

## SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
char c;
FILE *stream;

putchar (c)

fputc (c, stream)
FILE *stream;

putw (w, stream)
FILE *stream;
```

## DESCRIPTION

*Putc* appends the character *c* to the named output *stream*. It returns the character written.

*Putchar(c)* is defined as *putc(c, stdout)*.

*Fputc* behaves like *putc*, but is a genuine function rather than a macro. It may be used to save on object text.

*Putw* appends word (i.e. **int**) *w* to the output *stream*. It returns the word written. *Putw* neither assumes nor causes special alignment in the file.

The standard stream *stdout* is normally buffered if and only if the output does not refer to a terminal; this default may be changed by *setbuf*(3S). The standard stream *stderr* is by default unbuffered unconditionally, but use of *freopen*(3S) will cause it to become unbuffered; *setbuf*, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. See also *fflush*(3S).

## SEE ALSO

putc(3S), fopen(3S), getc(3S), puts(3S), printf(3S), fwrite(3S), ferror(3S)

## DIAGNOSTICS

These functions return the constant **EOF** upon error. Since this is a good integer, *ferror*(3S) should be used to detect *putw* errors.

## BUGS

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

## NAME

putc – buffered output

## SYNOPSIS

```
fcreat(file, iobuf)
char *file; struct buf *iobuf;

putc(c, iobuf)
int c;
struct buf *iobuf;

putw(w, iobuf);
int w;
struct buf *iobuf;

fflush(iobuf)
struct buf *iobuf;
```

## DESCRIPTION

*Fcreat* creates the given file (mode 666) and sets up the buffer *iobuf* (size 518 bytes); *putc* and *putw* write a byte or word respectively onto the file; *flush* forces the contents of the buffer to be written, but does not close the file. The format of the buffer is:

```
struct buf {
        int fildes;
        int nunused;
        char *nxtfree;
        char buff[512];
};
```

*Fcreat* returns –1 if file creation failed; none of the other routines returns error information.

Before terminating, a program should call *flush* to force out the last of the output (*fflush* from C).

The user must supply *iobuf*, which should begin on a word boundary.

To write a new file using the same buffer, it suffices to call [*f*]*flush*, close the file, and call *fcreat* again.

## SEE ALSO

creat(2), write(2), getc(3)

## NAME

putchar — write character

## SYNOPSIS

```
putchar (ch)
int ch;

flush ( )
```

## DESCRIPTION

*Putchar* writes out its argument and returns it unchanged. The low-order byte of the argument is always written; the high-order byte is written only if it is non-null. Unless other arrangements have been made, *putchar* writes in unbuffered fashion on the standard output file.

Associated with this routine is an external variable *fout* which has the structure of a buffer discussed under *putc:o*(3). If the file descriptor part of this structure (first word) is greater than 2, output via *putchar* is buffered. To achieve buffered output one may say, for example,

```
fout = dup(1);          or
fout = creat(...);
```

In such a case *flush* must be called before the program terminates in order to flush out the buffered output. *Flush* may be called at any time.

## SEE ALSO

putc:o(3)

## BUGS

The *fout* notion is kludgy.

## NAME

putpwent - write password file entry

## SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

## DESCRIPTION

*Putpwent* is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid*(3C) or *getpwnam*(3C)), *putpwuid* writes a line on the stream *f* which matches the format of **/etc/passwd**.

## DIAGNOSTICS

*Putpwent* returns nonzero if an error was detected during its operation, otherwise zero.

## NAME

puts, fputs − put a string on a stream

## SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

## DESCRIPTION

*Puts* copies the null-terminated string *s* to the standard output stream *stdout* and appends a new-line character.

*Fputs* copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

## DIAGNOSTICS

Both routines return EOF on error.

## SEE ALSO

fopen(3S), gets(3S), putc(3S), printf(3S), fwrite(3S), ferror(3S)

## NOTES

*Puts* appends a new-line, *fputs* does not.

NAME

qsort - quicker sort

SYNOPSIS

```
qsort (base, nel, width, compar)
char *base;
int nel, width;
int (*compar)( );
```

DESCRIPTION

*Qsort* is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1), strcmp(3C)

## NAME

rand, srand − random number generator

## SYNOPSIS

**srand (seed)**
**unsigned seed;**

**rand ( )**

## DESCRIPTION

*Rand* uses a multiplicative congruential random number generator with period $2^{32}$ to return successive pseudo-random numbers in the range from 0 to $2^{15}-1$. 32767

The generator is reinitialized by calling *srand* with 1 as argument. It can be set to a random starting point by calling *srand* with whatever you like as argument.

## NAME

reset − execute non-local goto

## SYNOPSIS

**setexit ( )**

**reset ( )**

## DESCRIPTION

These routines are useful for dealing with errors discovered in a low-level subroutine of a program.

*Setexit* is typically called just at the start of the main loop of a processing program. It stores certain parameters such as the call point and the stack level.

*Reset* is typically called after diagnosing an error in some subprocedure called from the main loop. When *reset* is called, it pops the stack appropriately and generates a non-local return from the last call to *setexit*.

It is erroneous, and generally disastrous, to call *reset* unless *setexit* has been called in a routine which is an ancestor of *reset*.

## NOTE

These routines have been replaced in the newer libraries with *setjmp*(3C) and *longjmp*(3C).

## BUGS

Only one non-local goto may be set up through this mechanism at a time.

## NAME

rmdir − remove directory

## SYNOPSIS

**rmdir (dirname)**
**char *dirname;**

## DESCRIPTION

*Rmdir* removes the directory specified by the partial or full pathname, *dirname*. *Dirname* is a string pointer. *Rmdir* checks the effective user id before doing anything. If the effective uid is not super user, control is returned to the caller with a **−2** return value. Thereafter no checking is done on any unlinks or closings. Hence if the process executing *rmdir* is aborted or killed in the process of doing an unlink the file system could result in a bad link count.

Return codes:

| | |
|---|---|
| **0** | successful *rmdir*. |
| **−1** | *dirname* not a directory or nonexistent. |
| **−2** | not allowed (could not open *dirname* or not super user). |
| **−3** | *dirname* not empty. |

*Rmdir* calls *close*(2), *getuid*(2), *open*(2), *read*(2), *stat*(2) and *unlink*(2) while removing *dirname*.

## SEE ALSO

mkdir:o(3C)

## BUGS

*Rmdir* should not require the effective user id to be super-user.

## NAME

scanf, fscanf, sscanf − formatted input conversion

## SYNOPSIS

```
#include <stdio.h>

scanf (format [ , pointer ] ... )
char *format;

fscanf (stream, format [ , pointer ] ... )
FILE *stream;
char *format;

sscanf (s, format [ , pointer ] ... )
char *s, *format;
```

## DESCRIPTION

*Scanf* reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or new-lines, which cause input to be read up to the next non-white-space character.
2. An ordinary character (not %) which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

**%** a single % is expected in the input at this point; no assignment is done.

**d** a decimal integer is expected; the corresponding argument should be an integer pointer.

**o** an octal integer is expected; the corresponding argument should be an integer pointer.

**x** a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

**s** a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which will be added. The input field is terminated by a space character or a new-line.

**c** a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next non-space character, try '%1s'. If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

**e** a floating point number is expected; the next field is converted accordingly and stored
**f** through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits possibly containing a

decimal point, followed by an optional exponent field consisting of an E or e followed by an optionally signed integer.

[ indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o** and **x** may be capitalized or preceded by **l** to indicate that a pointer to **long** rather than **int** is in the argument list. Similarly, the conversion characters **e** or **f** may be capitalized or preceded by **l** to indicate that a pointer to **double** rather than **float** is in the argument list. The character **h** will function similarly in the future to indicate **short** data items.

*Scanf* conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

*Scanf* returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

## EXAMPLES

The call:

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

**with the input line**

```
25  54.32E−1  thompson
```

will assign to *i* the value **25**, *x* the value **5.432**, and *name* will contain **thompson\0**. Or:

```
int i; float x; char name[50];
scanf ("%2d%f%*d%[1234567890]", &i, &x, name);
```

**with input:**

```
56789 0123 56a72
```

will assign **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* will return **a**.

## SEE ALSO

atof(3C), getc(3S), printf(3S)

## NOTE

Trailing white space (including a new-line) is left unread unless matched in the control string.

## DIAGNOSTICS

The *scanf* functions return EOF on end of input, and a short count for missing or illegal data items.

## BUGS

The success of literal matches and suppressed assignments is not directly determinable.

## NAME

setbuf – assign buffering to a stream

## SYNOPSIS

```
#include <stdio.h>

setbuf (stream, buf)
FILE *stream;
char *buf;
```

## DESCRIPTION

*Setbuf* is used after a stream has been opened but before it is read or written. It causes the character array *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from *malloc*(3C) upon the first *getc* or *putc*(3S) on the file, except that output streams directed to terminals, and the standard error stream *stderr* are normally not buffered.

## SEE ALSO

fopen(3S), getc(3S), putc(3S), malloc(3C)

## NAME

setjmp, longjmp − non-local goto

## SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

longjmp (env, val)
jmp_buf env;
```

## DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

*Setjmp* saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

*Longjmp* restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the function that invoked *setjmp*, which must not itself have returned in the interim. All accessible data have values as of the time *longjmp* was called.

## SEE ALSO

signal(2)

## NAME

sinh, cosh, tanh − hyperbolic functions

## SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

## DESCRIPTION

These functions compute the designated hyperbolic functions for real arguments.

## DIAGNOSTICS

*Sinh* and *cosh* return a huge value of appropriate sign when the correct value would overflow.

## NAME

sleep – stop execution for interval

## SYNOPSIS

**sleep (seconds)**
**unsigned seconds;**

## DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. *Sleep* is implemented by using *signal*(2) to catch the alarm clock signal, then using *alarm*(2) and *pause*(2) to set up and wait for the alarm to occur after the number of seconds specified by the argument. *Sleep* returns the number of seconds left of the sleep - always zero unless the sleep is interrupted by a signal.

*Sleep* and *alarm* interact intelligently, i.e. the C interface to sleep is intelligent enough to preserve the alarm signal vector and alarm timer while setting up a sleep. When a sleep is terminated, either by a signal or by the sleep time expiring, the alarm vector is set back to the value it had before the sleep and the alarm timer is set to the time it would have had taking into account the time elapsed while sleeping. Of course, if the alarm is scheduled to go off before the sleep time expires, *sleep* does not change the alarm vector or alarm timer. In that case, the time returned by *sleep* will be the sleep time less the alarm time; assuming, of course, that the alarm signal is the first signal to disturb *sleep*. Alarm timers will operate for the correct number of "real" seconds no matter how long signal processing routines take.

One consequence of the above scheme that is not adequately explained is as follows; suppose you set up to ignore alarms (*signal*(2)) set an alarm timer for 60 seconds, and then execute a 70 second *sleep*. After 60 seconds of time has elapsed, you will get a return from *sleep* of 10 seconds because the alarm time has expired. It might help in your understanding of *sleep* if you considered *sleep* to be a 'timed' pause function. *Pause* relinquishes the processor until the alarm timer expires; *sleep* relinquishes the processor until the alarm timer expires or until the requested sleep time expires, whichever comes first.

## SEE ALSO

sleep(1), alarm(2), pause(2), signal(2)

## BUGS

*Sleep* does not work correctly when called recursively. Only the last *sleep* called will return the correct value. The other *sleeps* will return the same value as the last *sleep*. If you don't particularly care what *sleep* returns, then *sleep* may be used recursively.

## NAME

ssignal, gsignal − software signals

## SYNOPSIS

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

## DESCRIPTION

*Ssignal* and *gsignal* implement a software facility similar to *signal*(2). This facility is used by the Standard C Library to enable the user to indicate the disposition of error conditions, and is also made available to the user for his own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. An *action* for a software signal is *established* by a call to *ssignal*, and a software signal is *raised* by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user defined) *action function* or one of the manifest constants **SIG_DFL** (default) or **SIG_IGN** (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns **SIG_DFL**.

*Gsignal* raises the signal identified by its argument, *sig*.

If an action function has been established for *sig*, then that action is reset to **SIG_DFL** and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.

If the action for *sig* is **SIG_IGN**, *gsignal* returns the value 1 and takes no other action.

If the action for *sig* is **SIG_DFL**, *gsignal* returns the value 0 and takes no other action.

If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

## NOTES

There are some additional signals with numbers outside the range 1 through 15 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

NAME

stdio − standard buffered input/output package

SYNOPSIS

**#include <stdio.h>**
**FILE *stdin, *stdout, *stderr;**

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are 3 open streams with constant pointers declared in the "include" file and associated with the standard open files:

| | |
|---|---|
| **stdin** | standard input file |
| **stdout** | standard output file |
| **stderr** | standard error file. |

A constant "pointer" NULL (0) designates the null stream.

An integer constant EOF (−1) is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

**#include <stdio.h>**

The functions and constants mentioned in the entries of sub-class 3S of this manual are declared in that "include" file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, and *fileno*.

SEE ALSO

open(2), close(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpnam(3S).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

## NAME

stdio − standard buffered input/output package

## SYNOPSIS

**#include <ostdio.h>**

**FILE *stdin;**
**FILE *stdout;**
**FILE *stderr;**

## DESCRIPTION

The functions described under subheading 3S constitute an efficient user-level buffering scheme. The in-line macros *getc*(3S) 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 a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. 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 |

In addition, there is a constant 'pointer' NULL (0) that designates the null stream.

An integer constant **EOF** (−1) is returned upon end of file or error by most integer functions that deal with streams.

Any routine using *occ* that uses the standard input/output package must include the header file of pertinent macro definitions this way:

**#include <ostdio.h>**

and must be loaded with a special library, obtained this way:

**cc ... −loS**

(Note that *occ* will convert any invocation of **-lS** to **-loS** automatically.) The functions and constants mentioned in subheading 3S are appropriately declared in the include file, and need no further declaration. The following 'functions' are implemented as macros; redeclaration of these names is perilous: *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *fileno*.

## SEE ALSO

open(2), close(2), read(2), write(2)

## DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. See individual function descriptions for possible error conditions.

Typical error conditions to watch for are a **FILE** pointer which has not been initialized with *fopen*, input (output) being attempted on an output (input) stream, or a **FILE** pointer which designates corrupt or otherwise unintelligible **FILE** data.

## NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr − string operations

## SYNOPSIS

```
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;
```

## DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

*Strcat* appends a copy of string *s2* to the end of string *s1*. *Strncat* copies at most *n* characters. Both return a pointer to the null-terminated result.

*Strcmp* compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *Strncmp* makes the same comparison but looks at at most *n* characters.

*Strcpy* copies string *s2* to *s1*, stopping after the null character has been moved. *Strncpy* copies exactly *n* characters, truncating or null-padding *s2;* the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

*Strlen* returns the number of non-null characters in *s*.

*Strchr* (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or NULL if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

## BUGS

*Strcmp* uses native character comparison, which is signed on PDP11s, unsigned on other machines.

## NAME

stty, gtty − set and retrieve the modes of a terminal

## SYNOPSIS

```
#include <sgtty.h>

int stty (fildes, arg)
int fildes;
struct sgttyb *arg;

int gtty (fildes, arg)
int fildes;
struct sgttyb *arg;
```

## DESCRIPTION

*Stty* and *gtty* are used to set and get various characteristics of a character device referred to by *fildes*. *Fildes* usually refers to a device associated with a typewriter, but may also refer to certain special devices such as named pipes. The second argument, *arg*, should be a pointer to the sgttyb structure defined in the include file **<sgtty.h>**.

## NOTE

*Stty* and *gtty* are now obsolete, having been replaced with the newer *ioctl* command. These routines merely call *ioctl* with the command requests **IOCSETP** and **TIOCGETP**. *Ioctl* should be used for all new code.

## SEE ALSO

stty(1), ioctl(2)

## DIAGNOSTICS

Zero is returned if the call was successful; a −1 return indicates an error.

## NAME

swab – swap bytes

## SYNOPSIS

```
swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

## DESCRIPTION

*Swab* copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP11s and other machines. *Nbytes* should be even.

## NAME

system − issue a shell command

## SYNOPSIS

```
#include <stdio.h>

int system (string)
char *string;
```

## DESCRIPTION

*System* causes the *string* to be given as input to the shell program specified in the environment variable **SHELL** (or **/bin/sh** if **SHELL** is not set) as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

## SEE ALSO

environ(7), exec(2), sh(1)

## DIAGNOSTICS

*System* stops if it can't execute *$SHELL* or */bin/sh*.

NAME

tell – get file offset

SYNOPSIS

**long tell (file)**
**int file;**

DESCRIPTION

*Tell* returns the current read/write pointer associated with the open file whose descriptor is specified as argument.

NOTE

*Tell* has been replaced by *lseek(file,0L,0)*, which should be used in all new code. *Tell* merely calls *lseek* with the proper arguments, and is provided only for compatibility with existing code.

SEE ALSO

lseek(3)

DIAGNOSTICS

–1 returned for an unknown file descriptor.

## NAME

tempfile − create a temporary file

## SYNOPSIS

```
#include <stdio.h>

FILE *tempfile ( )
```

## DESCRIPTION

*Tempfile* creates a temporary file, and returns a corresponding FILE pointer. Arrangements are made so that the file will automatically be deleted when the process using it terminates. The file is opened for update.

## SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), tmpnam(3S)

## BUGS

If called more than 17,576 times in a single process, *tmpnam* will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam* or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

tmpnam − create a name for a temporary file

SYNOPSIS

**#include <stdio.h>**

**char *tmpnam (s)**
**char *s;**

DESCRIPTION

*Tmpnam* generates a file name that can safely be used for a temporary file. If *(int)*s is zero, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *(int)*s is nonzero, *s* is assumed to be the address of an array of at least **L_tmpnam** bytes; *tmpnam* places its result in that array and returns *s* as its value.

*Tmpnam* generates a different file name each time it is called.

Files created using *tmpnam* and either *fopen* or *creat* are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink* to remove the file when its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C)

BUGS

If called more than 17,576 times in a single process, *tmpnam* will start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam* or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

## NAME

sin, cos, tan, asin, acos, atan, atan2 − trigonometric functions

## SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (x, y)
double x, y;
```

## DESCRIPTION

*Sin, cos,* and *tan* return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

*Asin* returns the arc sin in the range $-\pi/2$ to $\pi/2$.

*Acos* returns the arc cosine in the range 0 to $\pi$.

*Atan* returns the arc tangent of $x$ in the range $-\pi/2$ to $\pi/2$.

*Atan2* returns the arc tangent of $x/y$ in the range $-\pi$ to $\pi$.

## DIAGNOSTICS

Arguments of magnitude greater than 1 cause *asin* and *acos* to return value 0.

## NAME

ttyname, isatty – find name of a terminal

## SYNOPSIS

**char *ttyname (fildes)**

**int isatty (fildes)**

## DESCRIPTION

*Ttyname* returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

*Isatty* returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

## FILES

/dev/*

## SEE ALSO

ioctl(2)

## DIAGNOSTICS

*Ttyname* returns a null pointer (0) if *fildes* does not describe a terminal device in directory **/dev**.

## BUGS

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

## NAME

ttyslot − find the slot in the utmp file of the current user

## SYNOPSIS

**ttyslot ( )**

## DESCRIPTION

*Ttyslot* returns the index of the current user's entry in the *utmp* file. This is accomplished by actually scanning the file **/etc/lines** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2). A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors are associated with a terminal device.

## SEE ALSO

ttyname(3C)

## BUGS

*Ttyslot* returns 0 for errors while reading the *lines* file or if the file descriptors 0, 1 and 2 do not describe a terminal device.

NAME

ungetc - push character back into input stream

SYNOPSIS

**#include <stdio.h>**

**int ungetc (c, stream)**
**char c;**
**FILE *stream;**

DESCRIPTION

*Ungetc* pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *Ungetc* returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push **EOF** are rejected.

*Fseek*(3S) erases all memory of pushed back characters.

SEE ALSO

getc(3S), setbuf(3S), fseek(3S)

DIAGNOSTICS

*Ungetc* returns **EOF** if it can't push a character back.

## NAME

utindx, utline − access routines for utmp file

## SYNOPSIS

```
#include <utmp.h>

int utindx (id)
char id[3];

struct utmp *utline (id)
char id[3];
```

## DESCRIPTION

These routines provide access to the **utmp** file (normally **/etc/utmp**). This file contains user accounting information as maintained by **login**(1) and used by such commands as **who**(1). Both routines take a single argument which is a pointer to a string containing the two-character line id (from the **/etc/lines** file; see **lines**(6)) of interest.

*Utindx* returns the entry index of the most recent entry for the specified line ID. This index can then be used as the basis for direct access to the **utmp** file.

*Utline* returns the contents of the most recent entry for the specified line ID as a pointer to structure of the format specified in **/usr/include/utmp.h**:

```
/*              @(#)utmp.h      2.1             */
/*
 * Format of /etc/utmp and /usr/adm/wtmp
 */

#define UTMP_FILE "/etc/utmp"
#define WTMP_FILE "/etc/wtmp"

struct utmp {
                char            ut_name[8];             /* user id */
                char            ut_id[2];               /* line id */
                long            ut_time;                /* time on */
                int             ut_pid;                                 /* process id */
};
```

## SEE ALSO

login(1), utmp(5), lines(5), who(1)