## NAME

intro – introduction to functions and libraries

## DESCRIPTION

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

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). (For this reason the (3C) and (3S) sections together comprise one section of this manual.) The link editor *ld*(1) searches this library under the **–lc** option. A "shared library" version of *libc* can be searched using the **–lc_s** option, resulting in smaller *a.out*s. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

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

(3M) These functions constitute the Math Library, *libm*. They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there [see *math*(5)].

(3N) This contains sets of functions constituting the Network Services library. These sets provide protocol independent interfaces to networking services based on the service definitions of the OSI (Open Systems Interconnection) reference model. Application developers access the function sets that provide services at a particular level.

The function sets contained in the library are:

TRANSPORT INTERFACE (TI) - provide the services of the OSI Transport Layer. These services provide reliable end-to-end data transmission using the services of an underlying network. Applications written using the TI functions are independent of the underlying protocols. Declarations for these functions may be obtained from the **#include** file **<tiuser.h>**. The link editor *ld*(1) searches this library under the **–lnsl_s** option.

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

(3F) These functions constitute the FORTRAN intrinsic function library, *libF77*. These functions are automatically available to the FORTRAN programmer and require no special invocation of the compiler.

SYSTEM V/68 Release 3 provides support for the MC68881 floating point coprocessor chip. This support is in the form of optional MC68881 code generation by the *cc* compiler and by linking in the new libraries **libc881.a** and **libm881.a**. The user must explicitly link these libraries with the object code when selecting 881 support. Refer to *cc*(1) for more information about linking the libraries and setting the environment variable for floating point. Note that if the variable is selected but linking is with the default **libc.a** and **libm.a**, the floating point programs will not run correctly.

The documentation provided for **libc.a** and **libm.a** in sections 3C and 3M of the *SYSTEM V/68 User's Manual* describes the basic functionality of **libc881.a** and **libm881.a**. Additional information is provided in the following paragraphs and on the manual pages for *matherr*(3M), *math881*(3M) and *access881*(3C).

- The **libm881** routines expect rounding mode RN (round nearest) to be set on the MC68881 chip (this is the default chip setup). The user should not tamper with this rounding mode if the **libm881** routines are to be called. Otherwise, overflows may not return positive infinity as expected.
- When using **libc881**, the user should not set the rounding mode or rounding precision and should not capture 881 exceptions generated by C-compiler code. Doing so requires detailed knowledge of the C compiler's assumptions in doing code generation, and may lead to software that is not maintained easily.
- In general, C programmers writing 881-specific code (e.g., a signal handler for the fpu) are reminded that **fsave** and **frestore** are privileged instructions and the user must use the **sysm68k** call to retrieve the "exceptional operand" for an exception.

- Users writing their own signal handlers should note that multiple exceptions may be present and that the 881 status registers should be examined to detect this possible case.
- Users can handle 881 exceptions themselves by arranging to capture signal **SIGFPE** with the *signal*(2) system call (refer to *signal*(2) in the *SYSTEM V/68 User's Manual*). They can also look at the offending "exceptional operand" (which is retrieved with the privileged **FSAVE** instruction) by doing the *sysm68k* call with new command argument 3, which has the following form:

```
typedef long freg[3];
return_code = sysm68k(3,(freg)exc_op);
```

**where**

```
return_code = -1 ==> no 881 present
            = 0 ==> exceptional operand returned
```

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. The *null character* is a character with value 0, represented in the C language as '\0'. A *character array* is a sequence of characters. A *null-terminated character array* is a sequence of characters, the last of which is the *null character*. A *string* is a designation for a *null-terminated character array*. The *null string* is a character array containing only the null character. A NULL pointer is the value that is obtained by casting 0 into a pointer. The C language guarantees that this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. NULL is defined as 0 in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

Many groups of FORTRAN intrinsic functions have *generic* function names that do not require explicit or implicit type declaration. The type of the function will be determined by the type of its argument(s). For example, the generic function *max* will return an integer value if given integer arguments (*max0*), a real value if given real arguments (*amax1*), or a double-precision value if given double-precision arguments (*dmax1*).

**Netbuf** In the Network Services library, *netbuf* is a structure used in various Transport Interface (TI) functions to send and receive data and information. It contains the following members:

```
unsigned int maxlen;
unsigned int len;
char    *buf;
```

*Buf* points to a user input and/or output buffer. *Len* generally specifies the number of bytes contained in the buffer. If the structure is used for both input and output, the function will replace the user value of *len* on return.

*Maxlen* generally has significance only when *buf* is used to receive output from the TI function. In this case, it specifies the physical size of the buffer, the maximum value of *len* that can be set by the function. If *maxlen* is not large enough to hold the returned information, an TBUFOVFLW error will generally result. However, certain functions may return part of the data and not generate an error.

## FILES

LIBDIR usually /lib
LIBDIR/libc.a
LIBDIR/libc_s.a
LIBDIR/libm.a
LIBDIR/lib77.a
/shlib/libc_s
/shlib/libnsl_s (3N)
/usr/lib/libnsl_s.a (3N)

## SEE ALSO

ar(1), cc(1), ld(1), lint(1), nm(1), intro(2), stdio(3S), math(5).

## DIAGNOSTICS

Functions in the C and Math Libraries (3C and 3M) may return the conventional values **0** or ±**HUGE** (the largest-magnitude single-precision floating-point numbers; **HUGE** is defined in the <*math.h*> header file) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* [see *intro*(2)] is set to the value EDOM or ERANGE.

## WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Section 2 (*System Calls*). If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for Sections 2, 3C, and 3S are checked automatically. Other definitions can be included by using the **–l** option. (For example, –l*m* includes definitions for Section 3M, the Math Library.) Use of *lint* is highly recommended.

# SUBROUTINES (3)

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

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

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

*a64l* takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

*a64l* scans the character string from left to right, decoding each character as a 6 bit Radix 64 number.

*l64a* takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

CAVEAT

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

## NAME

abort – generate an IOT fault

## SYNOPSIS

**int abort ( )**

## DESCRIPTION

*abort* does the work of *exit*(2), but instead of just exiting, *abort* causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all *stdio*(3S) streams are flushed prior to the signal being sent, and a core dump results.

*abort* returns the value of the *kill*(2) system call.

## SEE ALSO

sdb(1), exit(2), kill(2), signal(2).

## DIAGNOSTICS

If **SIGABRT** is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message "abort – core dumped" is written by the shell.

## NAME

abs – return integer absolute value

## SYNOPSIS

```
int abs (i)
int i;
```

## DESCRIPTION

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

## SEE ALSO

floor(3M).

## CAVEAT

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

## NAME

access881 – provide access to floating point chip

## SYNOPSIS

**long rd881_status()**
**long rd881_iaddr()**
**long rd881_control()**

**void wr881_control(**newvalue**);**

## DESCRIPTION

The *access881* routines provide access from C to the system registers for the MC68881 floating point chip. The first three routines are used to read the chip's %status, %iaddr, and %control registers. The last routine is used to update the %control register with a new value.

## SEE ALSO

intro(3).

## NAME

bsearch – binary search a sorted table

## SYNOPSIS

**#include <search.h>**

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

## DESCRIPTION

*bsearch* is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *Key* points to a datum instance to be sought in the table. *Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero as accordingly the first argument is to be considered less than, equal to, or greater than the second.

## EXAMPLE

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <search.h>

#defineTABSIZE      1000

struct node {                   /* these are stored in the table */
        char *string;
        int length;
};
struct node table[TABSIZE];     /* table to be searched */
        .
        .
        .
{
```

(3C) and (3S)

```
                struct node *node_ptr, node;
                int node_compare( );  /* routine to compare 2 nodes */
                char str_space[20];   /* space to read string into */
                .
                .
                .
                node.string = str_space;
                while (scanf("%s", node.string) != EOF) {
                        node_ptr = (struct node *)bsearch((char *)(&node),
                                   (char *)table, TABSIZE,
                                   sizeof(struct node), node_compare);
                        if (node_ptr != NULL) {
                                (void)printf("string = %20s, length = %d\n",
                                        node_ptr->string, node_ptr->length);
                        } else {
                                (void)printf("not found: %s\n", node.string);
                        }
                }
        }
        /*
                This routine compares two nodes based on an
                alphabetical ordering of the string field.
        */
        int
        node_compare(node1, node2)
        char *node1, *node2;
        {
                return (strcmp(
                                ((struct node *)node1)->string,
                                ((struct node *)node2)->string));
        }
```

(3C) and (3S)

## NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although *bsearch* is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

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

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

(3C)
and
(3S)

## NAME

clock – report CPU time used

## SYNOPSIS

**long clock ( )**

## DESCRIPTION

*clock* returns the amount of CPU time (in microseconds) used since the first call to *clock*. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed *wait*(2), *pclose*(3S), or *system*(3S).

Typically, the resolution of the clock is 16.667 milliseconds on VME-based computers (actually 1/HZ. See <sys/param.h>).

## SEE ALSO

times(2), wait(2), popen(3S), system(3S).

## BUGS

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

## NAME

conv: toupper, tolower, _toupper, _tolower, toascii – translate characters

## SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

## DESCRIPTION

*Toupper* and *tolower* have as domain the range of *getc*(3S): the integers from –1 through 255. If the argument of *toupper* represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of *tolower* represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

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

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

## SEE ALSO

ctype(3C), getc(3S).

## NAME

crypt, setkey, encrypt – generate hashing encryption

## SYNOPSIS

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

void setkey (key)
char *key;

void encrypt (block, ignored)
char *block;
int ignored;
```

## DESCRIPTION

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

*Key* is a user's typed password. *Salt* is a two-character string chosen from the set [a-zA-Z0-9./]; 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 *setkey* and *encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the hashing algorithm to encrypt the string *block* with the function *encrypt*.

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 to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *setkey*. *Ignored* is unused by *encrypt* but it must be present.

## SEE ALSO

getpass(3C), passwd(4).
login(1), passwd(1) in the *User's Reference Manual.*

## CAVEAT

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

## NAME

ctermid – generate file name for terminal

## SYNOPSIS

```
#include <stdio.h>
char *ctermid (s)
char *s;
```

## DESCRIPTION

*ctermid* generates the path name of the controlling terminal for the current process, and stores it in a string.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least **L_ctermid** elements; the path name is placed in this array and the value of *s* is returned. The constant **L_ctermid** is defined in the *<stdio.h>* header file.

## NOTES

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

## SEE ALSO

ttyname(3C).

## NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to string

## SYNOPSIS

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

char *ctime (clock)
time_t *clock;

struct tm *localtime (clock)
time_t *clock;

struct tm *gmtime (clock)
time_t *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

## DESCRIPTION

*ctime* converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1985\n\0
```

*Localtime* and *gmtime* return pointers to "tm" structures, described below. *Localtime* corrects for the time zone and possible Daylight Savings Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the operating system uses.

*Asctime* converts a "tm" structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the "tm" structure, are in the <*time.h*> header file. The structure declaration is:

```
struct tm {
        int tm_sec;          /* seconds (0 - 59) */
        int tm_min;          /* minutes (0 - 59) */
```

```
            int tm_hour;      /* hours (0 - 23) */
            int tm_mday;      /* day of month (1 - 31) */
            int tm_mon;       /* month of year (0 - 11) */
            int tm_year;      /* year – 1900 */
            int tm_wday;      /* day of week (Sunday = 0) */
            int tm_yday;      /* day of year (0 - 365) */
            int tm_isdst;
        };
```

*Tm_isdst* is non-zero if Daylight Savings Time is in effect.

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

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

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

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

Note that in most installations, **TZ** is set by default when the user logs on, to a value in the local /etc/profile file [see *profile*(4)].

## SEE ALSO

time(2), getenv(3C), profile(4), environ(5).

## CAVEAT

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

## NAME

ctype: isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

## SYNOPSIS

**#include <ctype.h>**

**int isalpha (c)**
**int c;**

. . .

## DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** [–1; see *stdio*(3S)].

| | |
|---|---|
| *isalpha* | *c* is a letter. |
| *isupper* | *c* is an upper-case letter. |
| *islower* | *c* is a lower-case letter. |
| *isdigit* | *c* is a digit [0-9]. |
| *isxdigit* | *c* is a hexadecimal digit [0-9], [A-F] or [a-f]. |
| *isalnum* | *c* is an alphanumeric (letter or digit). |
| *isspace* | *c* is a space, tab, carriage return, newline, vertical tab, or form-feed. |
| *ispunct* | *c* is a punctuation character (neither control nor alphanumeric). |
| *isprint* | *c* is a printing character, code 040 (space) through 0176 (tilde). |
| *isgraph* | *c* is a printing character, like *isprint* except false for space. |
| *iscntrl* | *c* is a delete character (0177) or an ordinary control character (less than 040). |
| *isascii* | *c* is an ASCII character, code less than 0200. |

## SEE ALSO

stdio(3S), ascii(5).

## DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

## NAME

cuserid – get character login name of the user

## SYNOPSIS

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

## DESCRIPTION

*cuserid* generates a character-string representation of the login name that the owner of the current process is logged in under. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

## DIAGNOSTICS

If the login name cannot be found, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character (\0) will be placed at *s[0]*.

## SEE ALSO

getlogin(3C), getpwent(3C).

(3C)
and
(3S)

## NAME

dial – establish an out-going terminal line connection

## SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

## DESCRIPTION

*dial* returns a file-descriptor for a terminal line open for read/write. The argument to *dial* is a CALL structure (defined in the <*dial.h*> header file).

When finished with the terminal line, the calling program must invoke *undial* to release the semaphore that has been set during the allocation of the terminal device.

The definition of CALL in the <*dial.h*> header file is:

```
typedef struct {
  struct termio *attr;   /* pointer to termio attribute struct */
  int   baud;            /* transmission data rate */
  int   speed;           /* 212A modem: low=300, high=1200 */
  char   *line;          /* device name for out-going line */
  char   *telno;         /* pointer to tel-no digits string */
  int   modem;           /* specify modem control for direct lines */
  char   *device;        /*Will hold the name of the device used
                           to make a connection */
  int   dev_len;         /* The length of the device used to make
                           connection */
} CALL;
```

The CALL element *speed* is intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem will transmit at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per secound only. The CALL element *baud* is for the desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if **speed** set to 1200 **baud** must be set to high (1200).

If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the CALL structure. Legal values for such terminal device names are kept in the *Devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *Devices* file.

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Such numbers may consist only of symbols described on the *acu*(7). The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure.

The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL element *device* is used to hold the device name (cul..) that establishes the connection.

The CALL element *dev_len* is the length of the device name that is copied into the array device.

## FILES

/usr/lib/uucp/Devices
/usr/spool/locks/LCK..*tty-device*

## SEE ALSO

alarm(2), read(2), write(2).
termio(7) in the *System Administrator's Reference Manual*.
uucp(1C) in the *User's Reference Manual*.

## DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be returned. Mnemonics for these negative indices as listed here are defined in the <*dial.h*> header file.

| | | |
|---|---|---|
| INTRPT | –1 | /* interrupt occurred */ |
| D_HUNG | –2 | /* dialer hung (no return from write) */ |
| NO_ANS | –3 | /* no answer within 10 seconds */ |
| ILL_BD | –4 | /* illegal baud-rate */ |

```
A_PROB      -5    /* acu problem (open() failure) */
L_PROB      -6    /* line problem (open() failure) */
NO_Ldv      -7    /* can't open LDEVS file */
DV_NT_A     -8    /* requested device not available */
DV_NT_K     -9    /* requested device not known */
NO_BD_A     -10   /* no device available at requested baud */
NO_BD_K     -11   /* no device known at requested baud */
```

## WARNINGS

The *dial* (3C) library function is not compatible with Basic Networking Utilities on SYSTEM V/68 Release 3.

Including the **<dial.h>** header file automatically includes the **<termio.h>** header file.

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

## BUGS

An *alarm*(2) system call for 3600 seconds is made (and caught) within the *dial* module for the purpose of "touching" the *LCK..* file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1C) may simply delete the *LCK..* entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a *read*(2) or *write*(2) system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for **(errno==EINTR)**, and the *read* possibly reissued.

## NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

## SYNOPSIS

**double drand48 ( )**

**double erand48 (xsubi)**
**unsigned short xsubi[3];**

**long lrand48 ( )**

**long nrand48 (xsubi)**
**unsigned short xsubi[3];**

**long mrand48 ( )**

**long jrand48 (xsubi)**
**unsigned short xsubi[3];**

**void srand48 (seedval)**
**long seedval;**

**unsigned short *seed48 (seed16v)**
**unsigned short seed16v[3];**

**void lcong48 (param)**
**unsigned short param[7];**

## DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions *lrand48* and *nrand48* return non-negative long integers uniformly distributed over the interval [0, $2^{31}$ ).

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval [-$2^{31}$, $2^{31}$ ).

Functions *srand48, seed48* and *lcong48* are initialization entry points, one of which should be invoked before either *drand48, lrand48* or *mrand48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *erand48, nrand48* and *jrand48* do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, $X_i$, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48* has been invoked, the multiplier value *a* and the addend value *c* are given by

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_{8}.$$

The value returned by any of the functions *drand48, erand48, lrand48, nrand48, mrand48* or *jrand48* is computed by first generating the next 48-bit $X_i$ in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of $X_i$ and transformed into the returned value.

The functions *drand48, lrand48* and *mrand48* store the last 48-bit $X_i$ generated in an internal buffer, and must be initialized prior to being invoked. The functions *erand48, nrand48* and *jrand48* require the calling program to provide storage for the successive $X_i$ values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of $X_i$ into the array and pass it as an argument. By using different arguments, functions *erand48, nrand48* and *jrand48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers, i.e., the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *srand48* sets the high-order 32 bits of $X_i$ to the 32 bits contained in its argument. The low-order 16 bits of $X_i$ are set to the arbitrary value $330E_{16}$.

The initializer function *seed48* sets the value of $X_i$ to the 48-bit value specified in the argument array. In addition, the previous value of $X_i$ is copied into a 48-bit internal buffer, used only by *seed48,* and a pointer to this buffer is the value returned by *seed48*. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last $X_i$ value, and then use this value to reinitialize via *seed48* when the program is restarted.

The initialization function *lcong48* allows the user to specify the initial $X_i$ , the multiplier value *a*, and the addend value *c*. Argument array elements *param[0-2]* specify $X_i$ , *param[3-5]* specify the multiplier *a*, and *param[6]* specifies the 16-bit addend *c*. After *lcong48* has been called, a subsequent call to either *srand48* or *seed48* will restore the "standard" multiplier and addend values, *a* and *c*, specified on the previous page.

NOTES

The source code for the portable version can be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* are replaced by the two new functions below.

**long irand48 (m)**
**unsigned short m;**

**long krand48 (xsubi, m)**
**unsigned short xsubi[3], m;**

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval [0, *m*-1 ]. delim off

SEE ALSO

rand(3C).

## NAME

dup2 – duplicate an open file descriptor

## SYNOPSIS

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

## DESCRIPTION

*Fildes* is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than NOFILES. *dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

*dup2* will fail if one or more of the following are true:

| | |
|---|---|
| [EBADF] | *Fildes* is not a valid open file descriptor. |
| [EMFILE] | NOFILES file descriptors are currently open. |

## SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

## DIAGNOSTICS

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

(3C)
and
(3S)

## NAME

ecvt, fcvt, gcvt – convert floating-point number to string

## SYNOPSIS

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

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

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

## DESCRIPTION

*ecvt* converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. 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). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

*Fcvt* is identical to *ecvt,* except that the correct digit has been rounded for printf "%f" (FORTRAN F-format) output of the number of digits specified by *ndigit*.

*Gcvt* converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

## SEE ALSO

printf(3S).

## BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

## NAME

end, etext, edata – last locations in program

## SYNOPSIS

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

## DESCRIPTION

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

When execution begins, the program break (the first location beyond the data) coincides with *end*, but the program break may be reset by the routines of *brk*(2), *malloc*(3C), standard input/output [*stdio*(3S)], the profile (**–p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk (char *)(0)** [see *brk*(2)].

## SEE ALSO

cc(1), brk(2), malloc(3C), stdio(3S).

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

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

DESCRIPTION

*fclose* causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

*fclose* is performed automatically for all open files upon calling *exit*(2).

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

SEE ALSO

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

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

## NAME

ferror, feof, clearerr, fileno – stream status inquiries

## SYNOPSIS

**#include <stdio.h>**

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

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

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

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

## DESCRIPTION

*ferror* returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero.

*Feof* returns non-zero when EOF has previously been detected reading the named input *stream*, otherwise zero.

*Clearerr* resets the error indicator and EOF indicator to zero on the named *stream*.

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

## NOTES

All these functions are implemented as macros; they cannot be declared or redeclared.

## SEE ALSO

open(2), fopen(3S), stdio(3S).

## NAME

fopen, freopen, fdopen – open a stream

## SYNOPSIS

```
#include <stdio.h>

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

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

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

## DESCRIPTION

*fopen* opens the file named by *filename* and associates a *stream* with it. *fopen* returns a pointer to the FILE structure associated with the *stream*.

*Filename* points to a character string that contains the name of the file to be opened.

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

| | |
|---|---|
| "r" | open for reading |
| "w" | truncate or create for writing |
| "a" | append; open for writing at end of file, or create for writing |
| "r+" | open for update (reading and writing) |
| "w+" | truncate or create for update |
| "a+" | append; open or create for update at end-of-file |

*Freopen* substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. *Freopen* returns a pointer to the FILE structure associated with *stream*.

*Freopen* is typically used to attach the preopened *streams* associated with **stdin, stdout** and **stderr** to other files.

*Fdopen* associates a *stream* with a file descriptor. File descriptors are obtained from *open, dup, creat,* or *pipe*(2), which open files but do not return pointers to a FILE structure *stream*. Streams are necessary input for many of the Section 3S library routines. The *type* of *stream* must agree with the mode of the open file.

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

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. *Fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

## SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), stdio(3S).

## DIAGNOSTICS

*fopen*, *fdopen*, and *freopen* return a NULL pointer on failure.

## NAME

fread, fwrite – binary input/output

## SYNOPSIS

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

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

int fwrite (ptr, size, nitems, stream)
char *ptr;
int nitems;
size_t size;
FILE *stream;
```

## DESCRIPTION

*fread* copies, into an array pointed to by *ptr, nitems* items of data from the named input *stream,* where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size. fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream,* or if *nitems* items have been read. *fread* leaves the file pointer in *stream,* if defined, pointing to the byte following the last byte read if there is one. *fread* does not change the contents of *stream.*

*fwrite* appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream. fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream. fwrite* does not change the contents of the array pointed to by *ptr.*

The argument *size* is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr.* If *ptr* points to a data type other than *char* it should be cast into a pointer to *char.*

## SEE ALSO

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

## DIAGNOSTICS

*fread* and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite.*

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

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

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

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

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) $x$ is in the range $0.5 \leq |x| < 1.0$, and the "exponent" $n$ is an integer. *frexp* returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

*Ldexp* returns the quantity $value * 2^{exp}$.

*Modf* returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If *ldexp* would cause overflow, ±HUGE (defined in **<math.h>** ) is returned (according to the sign of *value*), and *errno* is set to ERANGE.
If *ldexp* would cause underflow, zero is returned and *errno* is set to ERANGE.

## NAME

fseek, rewind, ftell – reposition a file pointer in a stream

## SYNOPSIS

```
#include <stdio.h>

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

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

## DESCRIPTION

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

*Rewind*(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), except that no value is returned.

*fseek* and *rewind* undo any effects of *ungetc*(3S).

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

*Ftell* returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

## SEE ALSO

lseek(2), fopen(3S), popen(3S), stdio(3S), ungetc(3S).

## DIAGNOSTICS

*fseek* returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen;* in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

## WARNING

Although on the SYSTEM V/68 system an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-SYSTEM-V/68 systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

## NAME

ftw – walk a file tree

## SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

## DESCRIPTION

*ftw* recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw* calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure [see *stat*(2)] containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *stat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

*ftw* visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero. If *fn* returns a nonzero value, *ftw* stops its tree traversal and returns whatever value was returned by *fn*. If *ftw* detects an error, it returns –1, and sets the error type in *errno*.

*ftw* uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. *Depth* must not be greater than the number of file descriptors currently available for use. *ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

## SEE ALSO

stat(2), malloc(3C).

## BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

## CAVEAT

*ftw* uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

## NAME

getc, getchar, fgetc, getw – get character or word from a stream

## SYNOPSIS

**#include <stdio.h>**

**int getc (stream)**
**FILE *stream;**

**int getchar ()**

**int fgetc (stream)**
**FILE *stream;**

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

## DESCRIPTION

*getc* returns the next character (i.e., byte) from the named input *stream*, as an integer. It also moves the file pointer, if defined, ahead one character in *stream*. *getchar* is defined as *getc(stdin)*. *getc* and *getchar* are macros.

*Fgetc* behaves like *getc*, but is a function rather than a macro. *Fgetc* runs more slowly than *getc*, but it takes less space per invocation and its name can be passed as an argument to a function.

*Getw* returns the next word (i.e., integer) from the named input *stream*. *Getw* increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. *Getw* assumes no special alignment in the file.

## SEE ALSO

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

## DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

## WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and then compared against the integer constant **EOF**, the comparison may never succeed, because sign-extension of a character on widening to integer is machine-dependent.

## CAVEATS

Because it is implemented as a macro, *getc* evaluates a *stream* argument more than once. In particular, **getc(*f++)** does not work sensibly. *Fgetc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

```
char *getcwd (buf, size)
char *buf;
int size;
```

DESCRIPTION

*getcwd* returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free.*

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3C), popen(3S).
pwd(1) in the *User's Reference Manual.*

DIAGNOSTICS

Returns NULL with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

## NAME

getenv – return value for environment name

## SYNOPSIS

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

## DESCRIPTION

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

## SEE ALSO

exec(2), putenv(3C), environ(5).

## NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

## SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;
```

## DESCRIPTION

*getgrent*, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the /etc/group file. Each line contains a "group" structure, defined in the <*grp.h*> header file.

```
struct group {
        char    *gr_name;   /* the name of the group */
        char    *gr_passwd; /* the encrypted group password */
        int     gr_gid;     /* the numerical group ID */
        char    **gr_mem;   /* vector of pointers to member names */
};
```

*getgrent* when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *Getgrgid* searches from the beginning of the file until a numerical group id matching *gid* is found and returns a pointer to the particular structure in which it was found. *Getgrnam* searches from the beginning of the file until a group name matching *name* is found and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

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.

*Fgetgrent* returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

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

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

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

## NAME

getlogin – get login name

## SYNOPSIS

**char *getlogin ( );**

## DESCRIPTION

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

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

## FILES

/etc/utmp

## SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

## DIAGNOSTICS

Returns the **NULL** pointer if *name* is not found.

## CAVEAT

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

## NAME

getopt – get option letter from argument vector

## SYNOPSIS

**int getopt (argc, argv, optstring)**
**int argc;**
**char **argv, *opstring;**

**extern char *optarg;**
**extern int optind, opterr;**

## DESCRIPTION

*getopt* returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard (see *intro*(1)). So all new commands will adhere to the command syntax standard, they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command.

*optstring* must contain the option letters the command using *getopt* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

**optarg** is set to point to the start of the option-argument on return from *getopt*.

*getopt* places in **optind** the *argv* index of the next argument to be processed. **optind** is external and is initialized to **1** before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **–1**. The special option "—" may be used to delimit the end of the options; when it is encountered, **–1** will be returned, and "—" will be skipped.

## DIAGNOSTICS

*getopt* prints an error message on standard error and returns a question mark (?) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting **opterr** to **0**.

## EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the option **o**, which requires an option-argument:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abo:")) != -1)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                (void)fprintf(stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], 4)) {
        .
```

(3C) and (3S)

```
          :
    }
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o "xxx z yy" file
cmd -a -b -o "xxx z yy" -- file
cmd -ab -o xxx,z,yy file
cmd -ab -o "xxx z yy" file
cmd -o xxx,z,yy -b -a file
```

SEE ALSO

getopts(1), intro(1) in the *User's Reference Manual.*

Changing the value of the variable **optind**, or calling *getopt* with different values of *argv*, may lead to unexpected results.

NAME

getpass – read a password

SYNOPSIS

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

DESCRIPTION

*getpass* reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a **NULL** pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

CAVEAT

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

## NAME

getpw – get name from UID

## SYNOPSIS

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

## DESCRIPTION

*getpw* searches the password file for a user id number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. *getpw* returns non-zero if *uid* cannot be found.

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

## FILES

/etc/passwd

## SEE ALSO

getpwent(3C), passwd(4).

## DIAGNOSTICS

*getpw* returns non-zero on error.

## WARNING

The above routine uses **<stdio.h>**, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

## NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

## SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;
```

## DESCRIPTION

*getpwent*, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the **/etc/passwd** file. Each line in the file contains a "passwd" structure, declared in the <*pwd.h*> header file:

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

This structure is declared in <*pwd.h*> so it is not necessary to redeclare it.

The fields have meanings described in *passwd*(4).

*getpwent* when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching

(3C) and (3S)

*uid* is found and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

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

*Fgetpwent* returns a pointer to the next passwd structure in the stream *f*, which matches the format of **/etc/passwd.**

FILES

/etc/passwd

SEE ALSO

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

DIAGNOSTICS

A NULL pointer is returned on **EOF** or error.

WARNING

The above routines use **<stdio.h>**, which causes them to increase the size of programs, not otherwise using standard I/O, more than might be expected.

CAVEAT

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

## NAME

gets, fgets – get a string from a stream

## SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

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

## DESCRIPTION

*gets* reads characters from the standard input stream, *stdin,* into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

*Fgets* reads characters from the *stream* into the array pointed to by *s*, until *n*–1 characters are read, or a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

## SEE ALSO

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

## DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

## NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

## SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;
```

## DESCRIPTION

*getutent*, *getutid* and *getutline* each return a pointer to a structure of the following type:

```
struct utmp {
        char       ut_user[8];        /* User login name */
        char       ut_id[4];          /* /etc/inittab id (usually line #) */
        char       ut_line[12];       /* device name (console, lnxx) */
        short      ut_pid;            /* process id */
        short      ut_type;           /* type of entry */
        struct     exit_status {
           short      e_termination;  /* Process termination status */
           short      e_exit;         /* Process exit status */
        } ut_exit;                    /* The exit status of a process
                                       * marked as DEAD_PROCESS. */
        time_t     ut_time;           /* time entry was made */
};
```

*getutent* reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

*getutid* searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is

RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

*getutline* searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

*Pututline* writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

*Setutent* resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

*Endutent* closes the currently open file.

*Utmpname* allows the user to change the name of the file examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *Utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

## FILES

/etc/utmp
/etc/wtmp

## SEE ALSO

ttyslot(3C), utmp(4).

## DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

## NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

## NAME

hsearch, hcreate, hdestroy – manage hash search tables

## SYNOPSIS

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

## DESCRIPTION

*hsearch* is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *Item* is a structure of type ENTRY (defined in the *<search.h>* header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *Action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

*Hcreate* allocates sufficient space for the table, and must be called before *hsearch* is used. *Nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

*Hdestroy* destroys the search table, and may be followed by another call to *hcreate*.

## NOTES

*hsearch* uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

| | |
|---|---|
| DIV | Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm. |
| USCR | Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* [see *string*(3C)]. |
| CHAINED | Use a linked list to resolve collisions. If this option is selected, the following other options become available. |

| | |
|---|---|
| START | Place new entries at the beginning of the linked list (default is at the end). |
| SORTUP | Keep the linked list sorted by key in ascending order. |
| SORTDOWN | Keep the linked list sorted by key in descending order. |

Additionally, there are preprocessor flags for obtaining debugging printout (–DDEBUG) and for including a test driver in the calling routine (–DDRIVER). The source code should be consulted for further details.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

```
#include <stdio.h>
#include <search.h>

struct info {           /* this is the info stored in the table */
        int age, room; /* other than the key. */
};
#define NUM_EMPL     5000     /* # of elements in search table */

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
```

```
                    /* next avail space in info_space */
                    struct info *info_ptr = info_space;
                    ENTRY item, *found_item, *hsearch( );
                    /* name to look for in table */
                    char name_to_find[30];
                    int i = 0;

                    /* create table */
                    (void) hcreate(NUM_EMPL);
                    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                          &info_ptr->room) != EOF && i++ < NUM_EMPL) {
                          /* put info in structure, and structure in item */
                          item.key = str_ptr;
                          item.data = (char *)info_ptr;
                          str_ptr += strlen(str_ptr) + 1;
                          info_ptr++;
                          /* put item into table */
                          (void) hsearch(item, ENTER);
                    }

                    /* access table */
                    item.key = name_to_find;
                    while (scanf("%s", item.key) != EOF) {
                       if ((found_item = hsearch(item, FIND)) != NULL) {
                          /* if item is in the table */
                          (void)printf("found %s, age = %d, room = %d\n",
                                found_item->key,
                                ((struct info *)found_item->data)->age,
                                ((struct info *)found_item->data)->room);
                       } else {
                          (void)printf("no such employee %s\n",
                                name_to_find)
                       }
                    }
              }
```

(3C) and (3S)

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

DIAGNOSTICS

*hsearch* returns a NULL pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full.

*Hcreate* returns zero if it cannot allocate sufficient space for the table.

## WARNING

*hsearch* and *hcreate* use *malloc*(3C) to allocate space.

## CAVEAT

Only one hash search table may be active at any given time.

## NAME

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

## SYNOPSIS

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

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

## DESCRIPTION

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

*Ltol3* performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

## SEE ALSO

fs(4).

## CAVEAT

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

lockf – record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;
```

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

*Fildes* is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

*Function* is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define   F_ULOCK   0   /* Unlock a previously locked section */
#define   F_LOCK    1   /* Lock a section for exclusive use */
#define   F_TLOCK   2   /* Test and lock a section for exclusive use */
#define   F_TEST    3   /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

*Size* is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is zero, the section from the current offset through the largest file offset is locked (i.e., from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a –1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

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

[EACCES]
*Cmd* is F_TLOCK or F_TEST and the section is already locked by another process.

[EDEADLK]
*Cmd* is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.

[ECOMM]
*Fildes* is on a remote machine and the link to that machine is no longer active.

## SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2).

## DIAGNOSTICS

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

## WARNINGS

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

## NAME

lsearch, lfind – linear search and update

## SYNOPSIS

```
#include <stdio.h>
#include <search.h>

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

## DESCRIPTION

*lsearch* is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. **Key** points to the datum to be sought in the table. **Base** points to the first element in the table. **Nelp** points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table. **Compar** is the name of the comparison function which the user must supply (*strcmp*, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

*Lfind* is the same as *lsearch* except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

## NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

## EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.h>
```

```
#define TABSIZE 50
#define ELSIZE 120

        char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
        unsigned nel = 0;
        int strcmp( );
        . . .
        while (fgets(line, ELSIZE, stdin) != NULL &&
           nel < TABSIZE)
                (void) lsearch(line, (char *)tab, &nel,
                        ELSIZE, strcmp);
        . . .
```

**SEE ALSO**

bsearch(3C), hsearch(3C), string(3C), tsearch(3C).

**DIAGNOSTICS**

If the searched for datum is found, both *lsearch* and *lfind* return a pointer to it. Otherwise, *lfind* returns NULL and *lsearch* returns a pointer to the newly added element.

**BUGS**

Undefined results can occur if there is not enough room in the table to add a new item.

## NAME

malloc, free, realloc, calloc – main memory allocator

## SYNOPSIS

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

void free (ptr)
char *ptr;

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

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

## DESCRIPTION

*malloc* and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

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

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

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

*Realloc* changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

*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

brk(2), malloc(3X).

DIAGNOSTICS

*malloc*, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTES

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

## NAME

memory: memccpy, memchr, memcmp, memcpy, memset – memory operations

## SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

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

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

char *memset (s, c, n)
char *s;
int c, n;
```

## DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

*Memccpy* copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It returns a pointer to the character after the copy of **c** in **s1**, or a NULL pointer if **c** was not found in the first **n** characters of **s2**.

*Memchr* returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur.

*Memcmp* compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**.

*Memcpy* copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

*Memset* sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

For user convenience, all these functions are declared in the optional <*memory.h*> header file.

## CAVEATS

*Memcmp* is implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high order bit set is not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

## NAME

mktemp – make a unique file name

## SYNOPSIS

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

## DESCRIPTION

*mktemp* replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing Xs; *mktemp* will replace the Xs 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), tmpfile(3S), tmpnam(3S).

## DIAGNOSTIC

*mktemp* will assign to *template* the NULL string if it cannot create a unique name.

## CAVEAT

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

## NAME

monitor – prepare execution profile

## SYNOPSIS

```
#include <mon.h>

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
WORD *buffer;
int bufsize, nfunc;
```

## DESCRIPTION

An executable program created by **cc –p** automatically includes calls for *monitor* with default parameters; *monitor* need not 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* WORDs (defined in the <*mon.h*> header file). *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*. *Lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **–p** of *cc*(1) are recorded.

(3C) and (3S)

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

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

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

To stop execution monitoring and write the results, use

```
monitor ((int (*)())0, 0, 0, 0, 0);
```

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

The name of the file written by *monitor* is controlled by the environment variable PROFDIR. If PROFDIR does not exist, "mon.out" is created in the current directory. If PROFDIR exists but has no value, *monitor* does not do any profiling and creates no output file. Otherwise, the value of PROFDIR is used as the name of the directory in which to create the output file. If PROFDIR is *dirname*, then the file written is "*dirname/pid*.mon.out" where *pid* is the program's process id. (When *monitor* is called automatically by compiling via **cc -p**, the file created is "*dirname/pid.progname*" where *progname* is the name of the program.)

FILES

mon.out

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

BUGS

The "*dirname/pid*.mon.out" form does not work; the "*dirname/pid.progname*" form (automatically called via **cc -p**) does work.

(3C)
and
(3S)

NAME

nlist – get entries from name list

SYNOPSIS

**#include <nlist.h>**

**int nlist (filename, nl)**
**char *filename;**
**struct nlist *nl;**

DESCRIPTION

*nlist* examines the name list in the executable file whose name is pointed to by *filename*, and selectively extracts a list of values and puts them in the array of nlist structures pointed to by *nl*. The name list *nl* consists of an array of structures containing names of variables, types and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. The type field will be set to 0 unless the file was compiled with the –g option. If the name is not found, both entries are set to 0. See *a.out*(4) for a discussion of the symbol table structure.

This function is useful for examining the system name list kept in the file /sysV68. In this way programs can obtain system addresses that are up to date.

NOTES

The <*nlist.h*> header file is automatically included by <*a.out.h*> for compatability. However, if the only information needed from <*a.out.h*> is for use of *nlist*, then including <*a.out.h*> is discouraged. If <*a.out.h*> is included, the line "#undef n_name" may need to follow it.

SEE ALSO

a.out(4).

DIAGNOSTICS

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

*nlist* returns –1 upon error; otherwise it returns 0.

(3C)
and
(3S)

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

**void perror (s)**
**char *s;**

**extern int errno;**

**extern char *sys_errlist[ ];**

**extern int sys_nerr;**

DESCRIPTION

*perror* produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a new-line. (However, if s="" the colon is not printed.) To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index into this table to get the message string without the new-line. *Sys_nerr* is the number of messages 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 pipe to/from a process

## SYNOPSIS

**#include <stdio.h>**

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

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

## DESCRIPTION

*popen* creates a pipe between the calling program and the command to be executed. The arguments to *popen* are pointers to null-terminated strings. *Command* consists of a shell command line. *Type* is an I/O mode, either **r** for reading or **w** for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is **w**, by writing to the file *stream*; and one can read from the standard output of the command, if the I/O mode is **r**, by reading from the file *stream*.

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.

## EXAMPLE

A typical call may be:

```
char *cmd = "ls *.c";
FILE *ptr;
if ((ptr = popen(cmd, "r")) != NULL)
     while (fgets(buf, n, ptr) != NULL)
           (void) printf("%s ",buf);
```

This will print in *stdout* [see *stdio* (3S)] all the file names in the current directory that have a ".c" suffix.

## SEE ALSO

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

## DIAGNOSTICS

*popen* returns a NULL pointer if files or processes cannot be created.

*Pclose* returns –1 if *stream* is not associated with a "*popen*ed" command.

WARNING

If the original and "*popen*ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush* [see *fclose*(3S)].

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

**#include <stdio.h>**

**int printf (format , arg . . . )**
**char *format;**

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

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

DESCRIPTION

*printf* places output on the standard output stream *stdout*. *Fprintf* places output on the named output *stream*. *Sprintf* places "output," followed by the null character (\0), in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

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

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

A *precision* that gives the minimum number of digits to appear for the **d, i, o, u, x,** or **X** conversions, the number of digits to appear after the decimal point for the **e, E,** and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

An optional **l** (ell) specifying that a following **d, i, o, u, x,** or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision or both may be indicated by an asterisk (*****) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted. A negative field width argument is taken as a '–' flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

**–** The result of the conversion will be left-justified within the field.

**+** The result of a signed conversion will always begin with a sign (+ or –).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

**#** This flag specifies that the value is to be converted to an "alternate form." For **c, d, i, s,** and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** or **X** conversion, a non-zero result will have 0x or 0X prefixed to it. For **e, E, f, g,** and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

**d,i,o,u,x,X**
: The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal, (**o**), decimal (**u**), or hexadecimal notation (**x** or **X**), respectively; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

**f** The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

**e,E** The float or double *arg* is converted in the style "[–]d.ddde±dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

**g,G** The float or double *arg* is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style **e** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

**c** The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

% Print a %; no argument is converted.

In printing floating point types (float and double), if the exponent is 0x7FF and the mantissa is not equal to zero, then the output is

[-]NaN0xdddddddd

where 0xdddddddd is the hexadecimal representation of the leftmost 32 bits of the mantissa. If the mantissa is zero, the output is

[±]inf.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

## 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 %i, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

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

## SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <stdio.h>

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

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

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

DESCRIPTION

*putc* writes the character *c* onto the output *stream* (at the position where the file pointer, if defined, is pointing). *putchar*(*c*) is defined as *putc*(*c, stdout*). *putc* and *putchar* are macros.

*Fputc* behaves like *putc*, but is a function rather than a macro. *Fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

*Putw* writes the word (i.e. integer) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. *Putw* neither assumes nor causes special alignment in the file.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

On success, these functions (with the exception of *putw*) each return the value they have written. [*Putw* returns *ferror (stream)*]. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot grow. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

CAVEATS

Because it is implemented as a macro, *putc* evaluates a *stream* argument more than once. In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

putenv – change or add value to environment

SYNOPSIS

**int putenv (string)**
**char *string;**

DESCRIPTION

*String* points to a string of the form *"name=value." putenv* makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

DIAGNOSTICS

*putenv* returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero.

WARNINGS

*putenv* manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*. However, *envp* (the third argument to *main*) is not changed.
This routine uses *malloc*(3C) to enlarge the environment.
After *putenv* is called, environmental variables are not in alphabetical order.
A potential error is to call *putenv* with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

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* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd.**

SEE ALSO

getpwent(3C).

DIAGNOSTICS

*putpwent* returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

## 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* writes the null-terminated string pointed to by *s* ,followed by a new-line character, to the standard output stream *stdout.*

*Fputs* writes the null-terminated string pointed to by *s* to the named output *stream*.

Neither function writes the terminating null character.

## SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S).

## DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

## NOTES

*puts* appends a new-line character while *fputs* does not.

## NAME

qsort – quicker sort

## SYNOPSIS

**void qsort ((char *) base, nel, sizeof (*base), compar)**
**unsigned nel;**
**int (*compar)( );**

## DESCRIPTION

*qsort* is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

*Base* points to the element at the base of the table. *Nel* is the number of elements in the table. *Compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

## NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.
The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.
The order in the output of two items which compare as equal is unpredictable.

## SEE ALSO

bsearch(3C), lsearch(3C), string(3C).
sort(1) in the *User's Reference Manual.*

(3C)
and
(3S)

## NAME

rand, srand – simple random-number generator

## SYNOPSIS

```
int rand ( )

void srand (seed)
unsigned seed;
```

## DESCRIPTION

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

*Srand* can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

## NOTES

The spectral properties of *rand* are limited. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

## SEE ALSO

drand48(3C).

(3C)
and
(3S)

## NAME

scanf, fscanf, sscanf – convert formatted input

## SYNOPSIS

```
#include <stdio.h>

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

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

int 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 results are undefined in there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

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

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, 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, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

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 *. The suppression of assignment provides a way of describing an input field which is to be skipped. 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. For all descriptors except "[" and "c", white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes 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. |
| u | an unsigned decimal integer is expected; the corresponding argument should be an unsigned 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. |
| i | an integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal. |
| n | stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed. |
| e,f,g | a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional +, –, or space, followed by an integer. |

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

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

**[** indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating \0, which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, **x** and **i** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

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

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0** . Or:

```
int i, j; float x; char name[50];
(void) scanf("%i%2d%f%*d %[0–9] ", &j, &i, &x, name);
```

with input:

```
011 56789 0123 56a72
```

will assign **9** to *j*, **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* [see *getc*(3S)] will return **a**. Or:

```
int i, j, s, e; char name[50];
(void) scanf("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

```
0x11 0xy johnson
```

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is *e* - *s* = 2 . The next call to *getchar* [see *getc*(3S)] will return a blank.

(3C) and (3S)

## SEE ALSO

getc(3S), printf(3S), stdio(3S), strtod(3C), strtol(3C).

## DIAGNOSTICS

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

## CAVEATS

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

NAME

setbuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

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

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

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

A constant **BUFSIZ**, defined in the **<stdio.h>** header file, tells how big an array is needed:

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

*Setvbuf* may be used after a stream has been opened but before it is read or written. *Type* determines how *stream* will be buffered. Legal values for *type* (defined in stdio.h) are:

| | |
|---|---|
| _IOFBF | causes input/output to be fully buffered. |
| _IOLBF | causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested. |
| _IONBF | causes input/output to be completely unbuffered. |

If *buf* is not the **NULL** pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *Size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **<stdio.h>** is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

SEE ALSO

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

## DIAGNOSTICS

If an illegal value for *type* or *size* is provided, *setvbuf* returns a non-zero value. Otherwise, the value returned will be zero.

## NOTES

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

## NAME

setjmp, longjmp – non-local goto

## SYNOPSIS

**#include <setjmp.h>**

**int setjmp (env)**
**jmp_buf env;**

**void longjmp (env, val)**
**jmp_buf env;**
**int val;**

## DESCRIPTION

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

*setjmp* saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by *longjmp*. It returns the value 0.

*Longjmp* restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* (which must not itself have returned in the interim) had just returned the value *val*. *Longjmp* cannot cause *setjmp* to return the value 0. If *longjmp* is invoked with a second argument of 0, *setjmp* will return 1. At the time of the second return from *setjmp*, all accessible data have values as of the time *longjmp* is called. However, global variables will have the expected values, i.e. those as of the time of the *longjmp* (see example).

## EXAMPLE

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
        void exit();

        if(setjmp(env) != 0) {
                (void) printf("value of i on 2nd return from setjmp: %d\n", i);
                exit(0);
        }
        (void) printf("value of i on 1st return from setjmp: %d\n", i);
```

```
                i = 1;
                g();
                /*NOTREACHED*/
        }

        g()
        {
                longjmp(env, 1);
                /*NOTREACHED*/
        }
```

If the a.out resulting from this C language code is run, the output will be:

```
                value of i on 1st return from setjmp:0

                value of i on 2nd return from setjmp:1
```

## SEE ALSO

signal(2).

## WARNING

If *longjmp* is called even though *env* was never primed by a call to *setjmp*, or when the last such call was in a function which has since returned, absolute chaos is guaranteed.

## BUGS

The values of the registers on the second return from *setjmp* are the register values at the time of the first call to *setjmp*, not those at the time of the *longjmp*. This means that variables in a given function may behave differently in the presence of *setjmp*, depending on whether they are register or stack variables.

## NAME

sleep – suspend execution for interval

## SYNOPSIS

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

## DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

## SEE ALSO

alarm(2), pause(2), signal(2).

## NAME

ssignal, gsignal – software signals

## SYNOPSIS

```
#include <signal.h>

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

int gsignal (sig)
int sig;
```

## DESCRIPTION

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

Software signals made available to users are associated with integers in the inclusive range 1 through 16. A call to *ssignal* associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, 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.

## SEE ALSO

signal(2), sigset(2).

## NOTES

There are some additional signals with numbers outside the range 1 through 16 which are used by the Standard C Library to indicate error conditions. Thus, some signal numbers outside the range 1 through 16 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* and *putchar*, and the higher-level routines *fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, gets, getw, printf, puts, putw,* and *scanf* all use or act as if they 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 three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

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

A constant **NULL** (0) designates a nonexistent pointer.

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).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

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 header 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, ferror, feof, clearerr,* and *fileno*.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream *stderr* is by default unbuffered, but use of *freopen* [see *fopen*(3S)] will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). *Setbuf*(3S) or *setvbuf*() in *setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

open(2), close(2), lseek(2), pipe(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), tmpfile(3S), tmpnam(3S), ungetc(3S).

DIAGNOSTICS

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

NAME

stdipc: ftok – standard interprocess communication package

SYNOPSIS

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

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

*Ftok* returns a key based on *path* and *id* that is usable in subsequent *msgget*, *semget*, and *shmget* system calls. *Path* must be the path name of an existing file that is accessible to the process. *Id* is a character which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

*Ftok* returns **(key_t) –1** if *path* does not exist or if it is not accessible to the process.

WARNING

If the file whose *path* is passed to *ftok* is removed when keys still refer to the file, future calls to *ftok* with the same *path* and *id* will return an error. If the same file is recreated, then *ftok* is likely to return a different key than it did the original time it was called.

## NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

## SYNOPSIS

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

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

char *strdup (s1)
char *s1;

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

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

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

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

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

int strlen (s)
char *s;

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

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

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

int strspn (s1, s2)
char *s1, *s2;
```

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

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

DESCRIPTION

The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat, strncat, strcpy,* and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

*Strcat* appends a copy of string **s2** to the end of string **s1**.

*Strdup* returns a pointer to a new string which is a duplicate of the string pointed to by **s1**. The space for the new string is obtained using *malloc*(3C). If the new string can not be created, null is returned.

*Strncat* appends at most **n** characters. Each returns a pointer to the null-terminated result.

*Strcmp* compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater 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 copied. *Strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more. Each function returns **s1**.

*Strlen* returns the number of characters in **s**, not including the terminating null character.

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

*Strpbrk* returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a NULL pointer if no character from **s2** exists in **s1**.

*Strspn* (*strcspn*) returns the length of the initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

*Strtok* considers the string **s1** to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string **s2**. The first call (with pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into

**s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

For user convenience, all these functions are declared in the optional <*string.h*> header file.

## SEE ALSO

malloc(3C), malloc(3X).

## CAVEATS

*Strcmp* and *strncmp* are implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high-order bit set not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

## NAME

strtod, atof – convert string to double-precision number

## SYNOPSIS

```
double strtod (str, ptr)
char *str, **ptr;

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

## DESCRIPTION

*strtod* returns as a double-precision floating-point number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

*strtod* recognizes an optional string of "white-space" characters [as defined by *isspace* in *ctype*(3C)], then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, **ptr* is set to *str*, and zero is returned.

*Atof(str)* is equivalent to *strtod(str, (char **)NULL)*.

## SEE ALSO

ctype(3C), scanf(3S), strtol(3C).

## DIAGNOSTICS

If the correct value would cause overflow, ±HUGE (as defined in **<math.h>**) is returned (according to the sign of the value), and *errno* is set to ERANGE.
If the correct value would cause underflow, zero is returned and *errno* is set to ERANGE.

## NAME

strtol, atol, atoi – convert string to integer

## SYNOPSIS

```
long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;
```

## DESCRIPTION

*strtol* returns as a long integer the value represented by the character string pointed to by *str*. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters [as defined by *isspace* in *ctype*(3C)] are ignored.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

*Atol(str)* is equivalent to *strtol(str, (char **)NULL, 10)*.

*Atoi(str)* is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

## SEE ALSO

ctype(3C), scanf(3S), strtod(3C).

## CAVEAT

Overflow conditions are ignored.

## NAME

swab – swap bytes

## SYNOPSIS

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

## DESCRIPTION

*swab* copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*–1 instead. If *nbytes* is negative, *swab* does nothing.

## NAME

system – issue a shell command

## SYNOPSIS

```
#include <stdio.h>

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

## DESCRIPTION

*system* causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

## FILES

/bin/sh

## SEE ALSO

exec(2).
sh(1) in the *User's Reference Manual.*

## DIAGNOSTICS

*system* forks to create a child process that in turn exec's /bin/sh in order to execute *string*. If the fork or exec fails, *system* returns a negative value and sets *errno*.

## NAME

tmpfile – create a temporary file

## SYNOPSIS

**#include <stdio.h>**

**FILE *tmpfile ()**

## DESCRIPTION

*tmpfile* creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

## SEE ALSO

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

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions generate file names that can safely be used for a temporary file.

*tmpnam* always generates a file name using the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file. If *s* is NULL, *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 *s* is not NULL, it is assumed to be the address of an array of at least **L_tmpnam** bytes, where **L_tmpnam** is a constant defined in *<stdio.h>*; *tmpnam* places its result in that array and returns *s*.

*Tempnam* allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string that is not a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that directory is not accessible, /tmp will be used as a last resort. This entire sequence can be up-staged by providing an environment variable TMPDIR in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.

*Tempnam* uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* [see *malloc*(3C)]. If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

## NOTES

These functions generate a different file name each time they are called.

Files created using these functions and either *fopen*(3S) or *creat*(2) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when its use is ended.

## SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

## CAVEATS

If called more than 17,576 times in a single process, these functions 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 these functions or *mktemp*, and the file names are chosen to render duplication by other means unlikely.

## NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

## SYNOPSIS

```
#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );
```

## DESCRIPTION

*tsearch, tfind, tdelete,* and *twalk* are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

*tsearch* is used to build and access the tree. **Key** is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. **Rootp** points to a variable that points to the root of the tree. A NULL value for the variable pointed to by **rootp** denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

*Tdelete* deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be changed if the deleted node was the root of the tree. *Tdelete* returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

*Twalk* traverses a binary search tree. **Root** is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *Action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum* { *preorder, postorder, endorder, leaf* } *VISIT;* (defined in the *<search.h>* header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

**EXAMPLE**

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <stdio.h>

struct node {           /* pointers to these are stored in the tree */
        char *string;
        int length;
};
char string_space[10000];      /* space to store strings */
struct node nodes[500];        /* nodes to store */
struct node *root = NULL;      /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr−>string = strptr;
                nodeptr−>length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, (char **) &root,
                        node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr−>length + 1;
                nodeptr++;
        }
        twalk((char *)root, print_node);
}
/*
        This routine compares two nodes, based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
        return strcmp(((struct node *)node1)−>string,
        ((struct node *) node2)−>string);
}
/*
        This routine prints out a node, the first time
        twalk encounters it.
*/
void
print_node(node, order, level)
char **node;
VISIT order;
int level;
{
        if (order == preorder || order == leaf) {
                (void)printf("string = %20s, length = %d\n",
                        (*((struct node **)node))->string,
                        (*((struct node **)node))->length);
        }
}
```

**SEE ALSO**

bsearch(3C), hsearch(3C), lsearch(3C).

## DIAGNOSTICS

A NULL pointer is returned by *tsearch* if there is not enough space available to create a new node.

A NULL pointer is returned by *tfind* and *tdelete* if **rootp** is NULL on entry.

If the datum is found, both *tsearch* and *tfind* return a pointer to it. If not, *tfind* returns NULL, and *tsearch* returns a pointer to the inserted item.

## WARNINGS

The **root** argument to *twalk* is one level of indirection less than the **rootp** arguments to *tsearch* and *tdelete*.

There are two nomenclatures used to refer to the order in which tree nodes are visited. *tsearch* uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

## CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

## NAME

ttyname, isatty – find name of a terminal

## SYNOPSIS

```
char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;
```

## DESCRIPTION

*ttyname* returns a pointer to a string containing 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/*

## DIAGNOSTICS

*ttyname* returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

## CAVEAT

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

**int ttyslot ( )**

## DESCRIPTION

*ttyslot* returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

## FILES

/etc/inittab
/etc/utmp

## SEE ALSO

getut(3C), ttyname(3C).

## DIAGNOSTICS

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 is associated with a terminal device.

(3C)
and
(3S)

## NAME

ungetc – push character back into input stream

## SYNOPSIS

```
#include <stdio.h>

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

## DESCRIPTION

*ungetc* inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc*(3S) call on that *stream*. *ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

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

## SEE ALSO

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

## DIAGNOSTICS

*ungetc* returns EOF if it cannot insert the character.

## BUGS

When *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

(3C)
and
(3S)

## NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

## SYNOPSIS

```
#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 the same as *printf*, *fprintf*, and *sprintf* respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

## EXAMPLE

The following demonstrates the use of *vfprintf* to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *       error(function_name, format, arg1, arg2...);  */
/*VARARGS*/
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(stderr, fmt, args);
                va_end(args);
                (void)abort( );
        }
```

**SEE ALSO**

printf(3S), varargs(5).

## NAME

bessel: j0, j1, jn, y0, y1, yn – Bessel functions

## SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

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

## DESCRIPTION

*J0* and *j1* return Bessel functions of *x* of the first kind of orders 0 and 1 respectively. *Jn* returns the Bessel function of *x* of the first kind of order *n*.

*Y0* and *y1* return Bessel functions of *x* of the second kind of orders 0 and 1 respectively. *Yn* returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

## SEE ALSO

matherr(3M).

## DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return the value **–HUGE** and to set *errno* to **EDOM**. In addition, a message indicating DOMAIN error is printed on the standard error output.

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero and to set *errno* to **ERANGE**. In addition, a message indicating TLOSS error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

## NAME

erf, erfc – error function and complementary error function

## SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

## DESCRIPTION

*erf* returns the error function of *x*, defined as {2 over sqrt pi} int from 0 to x e sup {- t sup 2} dt .

*erfc*, which returns 1.0 – *erf(x)*, is provided because of the extreme loss of relative accuracy if *erf(x)* is called for large *x* and the result subtracted from 1.0 (e.g., for *x* = 5, 12 places are lost).

## SEE ALSO

exp(3M).

## NAME

exp, log, log10, pow, sqrt – exponential, logarithm, power, square root functions

## SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

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

double sqrt (x)
double x;
```

## DESCRIPTION

*exp* returns $e^x$.

*Log* returns the natural logarithm of $x$. The value of $x$ must be positive.

*Log10* returns the logarithm base ten of $x$. The value of $x$ must be positive.

*Pow* returns $x^y$. If $x$ is zero, $y$ must be positive. If $x$ is negative, $y$ must be an integer.

*Sqrt* returns the non-negative square root of $x$. The value of $x$ may not be negative.

## SEE ALSO

hypot(3M), matherr(3M), sinh(3M).

## DIAGNOSTICS

*exp* returns **HUGE** when the correct value would overflow, or 0 when the correct value would underflow, and sets *errno* to **ERANGE**.

*Log* and *log10* return **–HUGE** and set *errno* to **EDOM** when $x$ is non-positive. A message indicating DOMAIN error (or SING error when $x$ is 0) is printed on the standard error output.

*Pow* returns 0 and sets *errno* to **EDOM** when $x$ is 0 and $y$ is non-positive, or when $x$ is negative and $y$ is not an integer. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for *pow* would overflow or underflow, *pow* returns ±**HUGE** or 0 respectively, and sets *errno* to **ERANGE.**

*Sqrt* returns 0 and sets *errno* to **EDOM** when *x* is negative. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

(3M)

## NAME

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

## SYNOPSIS

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

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

double fabs (x)
double x;
```

## DESCRIPTION

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

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

*Fmod* returns the floating-point remainder of the division of $x$ by $y$: zero if $y$ is zero or if $x/y$ would overflow; otherwise the number $f$ with the same sign as $x$, such that $x = iy + f$ for some integer $i$, and $|f| < |y|$.

*Fabs* returns the absolute value of $x$, $|x|$.

## SEE ALSO

abs(3C).

(3M)

NAME

gamma – log gamma function

SYNOPSIS

**#include <math.h>**

**double gamma (x)**
**double x;**

**extern int signgam;**

DESCRIPTION

delim $$ *gamma* returns $ln ( | GAMMA ( ^ x ) | )$, where $GAMMA ( ^ x )$ is defined as $int from 0 to inf e sup { - t } t sup { x - 1 } dt$. The sign of GAMMA ( ^ x ) is returned in the external integer *signgam*. The argument *x* may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

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

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to return a range error, and is defined in the <*values.h*> header file.

SEE ALSO

exp(3M), matherr(3M), values(5).

DIAGNOSTICS

For non-negative integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

(3M)

NAME

hypot – Euclidean distance function

SYNOPSIS

**#include <math.h>**

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

DESCRIPTION

*hypot* returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

SEE ALSO

matherr(3M).

DIAGNOSTICS

When the correct value would overflow, *hypot* returns **HUGE** and sets *errno* to **ERANGE.**

These error-handling procedures may be changed with the function *matherr*(3M).

(3M)

## NAME

asin881, acos881, atan881, etox881, log881, logn881, mul881, sin881, cos881, sinh881, cosh881, sqrt881, tan881, tanh881 – floating point math functions

## SYNOPSIS

**unsigned int**
routine (x,y)
**double** x;
**double** *y;

## DESCRIPTION

The calling sequence for these functions is provided so that the routines may be accessed directly, rather than through use of the floating point math library. In the sequence, which is the same for all the routines, *x* is the input operand and *y* is the address to write the output operand to. If the operation requires two input operands, *y* also points to the second input operand. The routines return 881 status register results.

The following routine is used to see if either of a pair of input arguments is a NaN. It returns a non-zero value if one of the arguments is a NaN.

**int nan881**(a,b)
**double** a,b;

## SEE ALSO

matherr(3M), intro(3).

## NAME

matherr – error-handling function

## SYNOPSIS

**#include <math.h>**

**int matherr (x)**
**struct exception *x;**

## DESCRIPTION

*matherr* is invoked by functions in the Math Library when errors are detected. Users may define their own procedures for handling errors, by including a function named *matherr* in their programs. *matherr* must be of the form described above. When an error occurs, a pointer to the exception structure *x* will be passed to the user-supplied *matherr* function. This structure, which is defined in the *<math.h>* header file, is as follows:

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
};
```

The element *type* is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

| | |
|---|---|
| DOMAIN | argument domain error |
| SING | argument singularity |
| OVERFLOW | overflow range error |
| UNDERFLOW | underflow range error |
| TLOSS | total loss of significance |
| PLOSS | partial loss of significance |

The element *name* points to a string containing the name of the function that incurred the error. The variables *arg1* and *arg2* are the arguments with which the function was invoked. *Retval* is set to the default value that will be returned by the function unless the user's *matherr* sets it to a different value.

If the user's *matherr* function returns non-zero, no error message will be printed, and *errno* will not be set.

If *matherr* is not supplied by the user, the default error-handling procedures, described with the math functions involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, *errno* is set to EDOM or ERANGE and the program continues.

## EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
                /* change sqrt to return sqrt(-arg1), not 0 */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
                }
        case SING:
                /* all other domain or sing errors, print message and abort */
                fprintf(stderr, "domain error in %s\n", x->name);
                abort( );
        case PLOSS:
                /* print detailed error message */
                fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                        x->name, x->arg1, x->retval);
                return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
}
```

| DEFAULT ERROR HANDLING PROCEDURES | | | | | | |
|---|---|---|---|---|---|---|
| Types of Errors | | | | | | |
| type | DOMAIN | SING | OVERFLOW | UNDERFLOW | TLOSS | PLOSS |
| *errno* | EDOM | EDOM | ERANGE | ERANGE | ERANGE | ERANGE |
| BESSEL: | – | – | – | – | M,0 | * |
| y0, y1, yn (arg ≤) | M, –H | – | – | – | – | – |
| EXP: | – | – | H | 0 | – | – |
| POW: | | | | | | |
| neg **(non-int) | – | – | ±H | 0 | – | – |
| 0 ** (non-pos) | M, 0 | – | – | – | – | – |
| LOG, LOG10: | | | | | | |
| (arg < 0) | M,–H | – | – | – | – | – |
| (arg = 0) | – | M,–H | – | – | – | – |
| SQRT: | M, 0 | – | – | – | – | – |
| GAMMA: | – | M, H | H | – | – | – |
| HYPOT: | – | – | H | – | – | – |
| SINH: | – | – | ±H | – | – | – |
| COSH: | – | – | H | – | – | – |
| SIN, COS, TAN: | – | – | – | – | M, 0 | * |
| ASIN, ACOS, ATAN: | M,0 | – | – | – | – | – |

| ABBREVIATIONS | |
|---|---|
| * | As much as possible of the value is returned. |
| M | Message is printed (EDOM error). |
| H | HUGE is returned. |
| –H | –HUGE is returned. |
| ±H | HUGE or –HUGE is returned. |
| 0 | 0 is returned. |

NOTE: In addition to the errors listed in the table above, all **libm881** routines will return a **DOMAIN** error when an input to the routine is a NaN.

If input is positive infinity, the functions *log* and *log10* will return a "quiet" overflow condition.

(3M)

## 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

*sinh*, *cosh*, and *tanh* return, respectively, the hyberbolic sine, cosine and tangent of their argument.

## SEE ALSO

matherr(3M).

## DIAGNOSTICS

*sinh* and *cosh* return **HUGE** (and *sinh* may return **-HUGE** for negative *x*) when the correct value would overflow and set *errno* to **ERANGE**.

These error-handling procedures may be changed with the function *matherr*(3M).

(3M)

## NAME

trig: sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

## SYNOPSIS

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

## DESCRIPTION

*Sin*, *cos* and *tan* return respectively the sine, cosine and tangent of their argument, *x*, measured in radians.

*Asin* returns the arcsine of *x*, in the range $[-\pi/2,\pi/2]$.

*Acos* returns the arccosine of *x*, in the range $[0,\pi]$.

*Atan* returns the arctangent of *x*, in the range $[-\pi/2,\pi/2]$.

*Atan2* returns the arctangent of *y/x*, in the range $(-\pi,\pi]$, using the signs of both arguments to determine the quadrant of the return value.

## SEE ALSO

matherr(3M).

## DIAGNOSTICS

*Sin*, *cos*, and *tan* lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return zero when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, *errno* is set to **ERANGE.**

If the magnitude of the argument of *asin* or *acos* is greater than one, or if both arguments of *atan2* are zero, zero is returned and *errno* is set to EDOM. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the function *matherr*(3M).

## NAME

t_accept – accept a connect request

## SYNOPSIS

```
#include <tiuser.h>

int t_accept(fd, resfd, call)
int fd;
int resfd;
struct t_call *call;
```

## DESCRIPTION

This function is issued by a transport user to accept a connect request. *Fd* identifies the local transport endpoint where the connect indication arrived, *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. *Call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*Netbuf* is described in *intro*(3). In *call, addr* is the address of the caller, *opt* indicates any protocol-specific parameters associated with the connection, *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by *t_listen* that uniquely associates the response with a previously received connect indication.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. If the same endpoint is specified (i.e., *resfd=fd*), the connection can be accepted unless the following condition is true: The user has received other indications on that endpoint but has not responded to them (with *t_accept* or *t_snddis*). For this condition, *t_accept* will fail and set *t_errno* to TBADF.

If a different transport endpoint is specified (*resfd!=fd*), the endpoint must be bound to a protocol address and must be in the T_IDLE state [see *t_getstate*(3N)] before the *t_accept* is issued.

For both types of endpoints, *t_accept* will fail and set *t_errno* to TLOOK if there are indications (e.g., a connect or disconnect) waiting to be received on that endpoint.

The values of parameters specified by *opt* and the syntax of those values are protocol specific. The *udata* argument enables the called transport user to send user data to the caller and the amount of user data must not exceed the limits supported by the transport provider as returned by *t_open* or *t_getinfo*. If the *len* [see *netbuf* in *intro*(3)] field of *udata* is zero, no data will be sent to the caller.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint, or the user is illegally accepting a connection on the same transport endpoint on which the connect indication arrived. |
| [TOUTSTATE] | The function was issued in the wrong sequence on the transport endpoint referenced by *fd*, or the transport endpoint referred to by *resfd* is not in the T_IDLE state. |
| [TACCES] | The user does not have permission to accept a connection on the responding transport endpoint or use the specified options. |
| [TBADOPT] | The specified options were in an incorrect format or contained illegal information. |
| [TBADDATA] | The amount of user data specified was not within the bounds allowed by the transport provider. |
| [TBADSEQ] | An invalid sequence number was specified. |
| [TLOOK] | An asynchronous event has occurred on the transport endpoint referenced by *fd* and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_connect(3N), t_getstate(3N), t_listen(3N), t_open(3N), t_rcvconnect(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

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

## NAME

t_alloc – allocate a library structure

## SYNOPSIS

```
#include <tiuser.h>

char *t_alloc(fd, struct_type, fields)
int fd;
int struct_type;
int fields;
```

## DESCRIPTION

The *t_alloc* function dynamically allocates memory for the various transport function argument structures as specified below. This function will allocate memory for the specified structure, and will also allocate memory for buffers referenced by the structure.

The structure to allocate is specified by *struct_type,* and can be one of the following:

| | |
|---|---|
| T_BIND | struct t_bind |
| T_CALL | struct t_call |
| T_OPTMGMT | struct t_optmgmt |
| T_DIS | struct t_discon |
| T_UNITDATA | struct t_unitdata |
| T_UDERROR | struct t_uderr |
| T_INFO | struct t_info |

where each of these structures may subsequently be used as an argument to one or more transport functions.

Each of the above structures, except T_INFO, contains at least one field of type *struct netbuf*. *Netbuf* is described in *intro*(3). For each field of this type, the user may specify that the buffer for that field should be allocated as well. The *fields* argument specifies this option, where the argument is the bitwise-OR of any of the following:

T_ADDR The *addr* field of the *t_bind, t_call, t_unitdata,* or *t_uderr* structures.

| | |
|---|---|
| T_OPT | The *opt* field of the *t_optmgmt*, *t_call*, *t_unitdata*, or *t_uderr* structures. |
| T_UDATA | The *udata* field of the *t_call*, *t_discon*, or *t_unitdata* structures. |
| T_ALL | All relevant fields of the given structure. |

For each field specified in *fields*, *t_alloc* will allocate memory for the buffer associated with the field, and initialize the *buf* pointer and *maxlen* [see *netbuf* in *intro*(3) for description of *buf* and *maxlen*] field accordingly. The length of the buffer allocated will be based on the same size information that is returned to the user on *t_open* and *t_getinfo*. Thus, *fd* must refer to the transport endpoint through which the newly allocated structure will be passed, so that the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 (see *t_open* or *t_getinfo*), *t_alloc* will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to TSYSERR and *errno* to EINVAL. For any field not specified in *fields*, *buf* will be set to NULL and *maxlen* will be set to zero.

Use of *t_alloc* to allocate structures will help ensure the compatibility of user programs with future releases of the transport interface.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_free(3N), t_getinfo(3N), t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

On successful completion, *t_alloc* returns a pointer to the newly allocated structure. On failure, NULL is returned.

## NAME

t_bind – bind an address to a transport endpoint

## SYNOPSIS

**#include <tiuser.h>**

```
int t_bind(fd, req, ret)
int fd;
struct t_bind *req;
struct t_bind *ret;
```

## DESCRIPTION

This function associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin accepting or requesting connections on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* arguments point to a *t_bind* structure containing the following members:

```
struct netbuf addr;
unsigned qlen;
```

*Netbuf* is described in *intro*(3). The *addr* field of the *t_bind* structure specifies a protocol address and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

*Req* is used to request that an address, represented by the *netbuf* structure, be bound to the given transport endpoint. *Len* [see *netbuf* in *intro*(3); also for *buf* and *maxlen*] specifies the number of bytes in the address and *buf* points to the address buffer. *Maxlen* has no meaning for the *req* argument. On return, *ret* contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified by the user in *req*. In *ret*, the user specifies *maxlen* which is the maximum size of the address buffer and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error will result.

If the requested address is not available, or if no address is specified in *req* (the *len* field of *addr* in *req* is zero) the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. The user can compare the addresses in *req* and *ret* to determine whether the transport provider bound the transport endpoint to a different address than that requested.

*Req* may be NULL if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider must assign an address to the transport endpoint. Similarly, *ret* may be NULL if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to NULL for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The *qlen* field has meaning only when initializing a connection-mode service. It specifies the number of outstanding connect indications the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and may be changed if the transport provider cannot support the specified number of outstanding connect indications. On return, the *qlen* field in *ret* will contain the negotiated value.

This function allows more than one transport endpoint to be bound to the same protocol address (however, the transport provider must support this capability also), but it is not allowable to bind more than one protocol address to the same transport endpoint. If a user binds more than one transport endpoint to the same protocol address, only one endpoint can be used to listen for connect indications associated with that protocol address. In other words, only one *t_bind* for a given protocol address may specify a value of *qlen* greater than zero. In this way, the transport provider can identify which transport endpoint should be notified of an incoming connect indication. If a user attempts to bind a protocol address to a second transport endpoint with a value of *qlen* greater than zero, the transport provider will assign another address to be bound to that endpoint. If a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of that connection. No other transport endpoints may be bound for listening while that initial listening endpoint is in the data transfer phase. This will prevent more than one transport endpoint bound to the same protocol address from accepting connect indications.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TOUTSTATE] | The function was issued in the wrong sequence. |
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information. |
| [TNOADDR] | The transport provider could not allocate an address. |
| [TACCES] | The user does not have permission to use the specified address. |
| [TBUFOVFLW] | The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_open(3N), t_optmgmt(3N), t_unbind(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_bind* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_close – close a transport endpoint

## SYNOPSIS

```
#include <tiuser.h>

int t_close(fd)
int fd;
```

## DESCRIPTION

The *t_close* function informs the transport provider that the user is finished with the transport endpoint specified by *fd,* and frees any local library resources associated with the endpoint. In addition, *t_close* closes the file associated with the transport endpoint.

*t_close* should be called from the T_UNBND state [see *t_getstate* (3N)]. However, this function does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint will be freed automatically. In addition, *close*(2) will be issued for that file descriptor; the close will be abortive if no other process has that file open, and will break any transport connection that may be associated with that endpoint.

On failure, *t_errno* may be set to the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |

## SEE ALSO

t_getstate(3N), t_open(3N), t_unbind(3N).
*SYSTEM V/68 Programmer's Guide* .

## DIAGNOSTICS

*t_close* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_connect – establish a connection with another transport user

## SYNOPSIS

```
#include <tiuser.h>

int t_connect(fd, sndcall, rcvcall)
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;
```

## DESCRIPTION

This function enables a transport user to request a connection to the specified destination transport user. *Fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*Sndcall* specifies information needed by the transport provider to establish a connection and *rcvcall* specifies information that is associated with the newly established connection.

*Netbuf* is described in *intro*(3). In *sndcall, addr* specifies the protocol address of the destination transport user, *opt* presents any protocol-specific information that might be needed by the transport provider, *udata* points to optional user data that may be passed to the destination transport user during connection establishment, and *sequence* has no meaning for this function.

On return in *rcvcall, addr* returns the protocol address associated with the responding transport endpoint, *opt* presents any protocol-specific information associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during connection establishment, and *sequence* has no meaning for this function.

The *opt* argument implies no structure on the options that may be passed to the transport provider. The transport provider is free to specify the structure of any options passed to it. These options are specific to the underlying protocol of the transport provider. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

The *udata* argument enables the caller to pass user data to the destination transport user and receive user data from the destination user during connection establishment. However, the amount of user data must not exceed the limits supported by the transport provider as returned by *t_open* (3N) or *t_getinfo* (3N). If the *len* [see *netbuf* in *intro*(3)] field of *udata* is zero in *sndcall,* no data will be sent to the destination transport user.

On return, the *addr, opt,* and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* [see *netbuf* in *intro*(3)] field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be NULL, in which case no information is given to the user on return from *t_connect.*

By default, *t_connect* executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (i.e. return value of zero) indicates that the requested connection has been established. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_connect* executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with *t_errno* set to TNODATA to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TOUTSTATE] | The function was issued in the wrong sequence. |
| [TNODATA] | O_NDELAY was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user. |
| [TBADADDR] | The specified protocol address was in an incorrect format or contained illegal information. |

| | |
|---|---|
| [TBADOPT] | The specified protocol options were in an incorrect format or contained illegal information. |
| [TBADDATA] | The amount of user data specified was not within the bounds allowed by the transport provider. |
| [TACCES] | The user does not have permission to use the specified address or options. |
| [TBUFOVFLW] | The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the connect indication information to be returned in *rcvcall* is discarded. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_accept(3N), t_getinfo(3N), t_listen(3N), t_open(3N), t_optmgmt(3N), t_rcvconnect(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_connect* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_error – produce error message

## SYNOPSIS

```
#include <tiuser.h>

void t_error(errmsg)
char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

## DESCRIPTION

The *t_error* function produces a message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error. *t_error* prints the user-supplied error message followed by a colon and a standard error message for the current error defined in *t_errno*. To simplify variant formatting of messages, the array of message strings *t_errlist* is provided; *t_errno* can be used as an index in this table to get the message string without the newline. *T_nerr* is the largest message number provided for in the *t_errlist* table.

*T_errno* is only set when an error occurs and is not cleared on successful calls.

## EXAMPLE

If a *t_connect* function fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2:  Incorrect transport address format
```

where "Incorrect transport address format" identifies the specific error that occurred, and "t_connect failed on fd2" tells the user which function failed on which transport endpoint.

## SEE ALSO

*SYSTEM V/68 Programmer's Guide*

## NAME

t_free – free a library structure

## SYNOPSIS

```
#include <tiuser.h>

int t_free(ptr, struct_type)
char *ptr;
int struct_type;
```

## DESCRIPTION

The *t_free* function frees memory previously allocated by *t_alloc*. This function will free memory for the specified structure, and will also free memory for buffers referenced by the structure.

*Ptr* points to one of the six structure types described for *t_alloc*, and *struct_type* identifies the type of that structure which can be one of the following:

| | |
|---|---|
| T_BIND | struct t_bind |
| T_CALL | struct t_call |
| T_OPTMGMT | struct t_optmgmt |
| T_DIS | struct t_discon |
| T_UNITDATA | struct t_unitdata |
| T_UDERROR | struct t_uderr |
| T_INFO | struct t_info |

where each of these structures is used as an argument to one or more transport functions.

*t_free* will check the *addr*, *opt*, and *udata* fields of the given structure (as appropriate), and free the buffers pointed to by the *buf* field of the *netbuf* [see *intro*(3)] structure. If *buf* is NULL, *t_free* will not attempt to free memory. After all buffers are freed, *t_free* will free the memory associated with the structure pointed to by *ptr*.

Undefined results will occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by *t_alloc*.

(3N)

On failure, *t_errno* may be set to the following:

| | |
|---|---|
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_alloc(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_free* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_getinfo – get protocol-specific service information

## SYNOPSIS

```
#include <tiuser.h>

int t_getinfo(fd, info)
int fd;
struct t_info *info;
```

## DESCRIPTION

This function returns the current characteristics of the underlying transport protocol associated with file descriptor *fd*. The *info* structure is used to return the same information returned by *t_open*. This function enables a transport user to access this information during any phase of communication.

This argument points to a *t_info* structure which contains the following members:

```
long addr;      /* max size of the transport protocol address */
long options;   /* max number of bytes of protocol-specific options */
long tsdu;      /* max size of a transport service data unit (TSDU) */
long etsdu;     /* max size of an expedited transport service data unit (ETSDU) *
long connect;   /* max amount of data allowed on connection establishment
                   functions */
long discon;    /* max amount of data allowed on t_snddis and t_rcvdis functions
long servtype;  /* service type supported by the transport provider */
```

The values of the fields have the following meanings:

| | |
|---|---|
| *addr* | A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses. |
| *options* | A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options. |

| | |
|---|---|
| *tsdu* | A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider. |
| *etsdu* | A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider. |
| *connect* | A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions. |
| *discon* | A value greater than or equal to zero specifies the maximum amount of data that may be associated with the *t_snddis* and *t_rcvdis* functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions. |
| *servtype* | This field specifies the service type supported by the transport provider, as described below. |

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc* function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of option negotiation, and *t_getinfo* enables a user to retrieve the current characteristics.

The *servtype* field of *info* may specify one of the following values on return:

| | |
|---|---|
| T_COTS | The transport provider supports a connection-mode service but does not support the optional orderly release facility. |
| T_COTS_ORD | The transport provider supports a connection-mode service with the optional orderly release facility. |
| T_CLTS | The transport provider supports a connectionless-mode service. For this service type, *t_open* will return -2 for *etsdu*, *connect*, and *discon*. |

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_getinfo* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_getstate – get the current state

## SYNOPSIS

**#include <tiuser.h>**

**int t_getstate(fd)**
**int fd;**

## DESCRIPTION

The *t_getstate* function returns the current state of the provider associated with the transport endpoint specified by *fd*.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TSTATECHNG] | The transport provider is undergoing a state change. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_getstate* returns the current state on successful completion and -1 on failure and *t_errno* is set to indicate the error. The current state may be one of the following:

| | |
|---|---|
| T_UNBND | unbound |
| T_IDLE | idle |
| T_OUTCON | outgoing connection pending |
| T_INCON | incoming connection pending |
| T_DATAXFER | data transfer |
| T_OUTREL | outgoing orderly release (waiting for an orderly release indication) |

| | |
|---|---|
| T_INREL | incoming orderly release (waiting for an orderly release request) |

If the provider is undergoing a state transition when *t_getstate* is called, the function will fail.

## NAME

t_listen – listen for a connect request

## SYNOPSIS

```
#include <tiuser.h>

int t_listen(fd, call)
int fd;
struct t_call *call;
```

## DESCRIPTION

This function listens for a connect request from a calling transport user. *Fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication. *Call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*Netbuf* is described in *intro*(3). In *call, addr* returns the protocol address of the calling transport user, *opt* returns protocol-specific parameters associated with the connect request, *udata* returns any user data sent by the caller on the connect request, and *sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr, opt,* and *udata* fields of *call,* the *maxlen* [see *netbuf* in *intro*(3)] field of each must be set before issuing the *t_listen* to indicate the maximum size of the buffer for each.

By default, *t_listen* executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_listen* executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets *t_errno* to TNODATA.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect indication information to be returned in *call* is discarded. |
| [TNODATA] | O_NDELAY was set, but no connect indications had been queued. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## CAVEATS

If a user issues *t_listen* in synchronous mode on a transport endpoint that was not bound for listening (i.e. *qlen* was zero on *t_bind*), the call will wait forever because no connect indications will arrive on that endpoint.

## SEE ALSO

intro(3), t_accept(3N), t_bind(3N), t_connect(3N), t_open(3N), t_rcvconnect(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_listen* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_look – look at the current event on a transport endpoint

## SYNOPSIS

```
#include <tiuser.h>

int t_look(fd)
int fd;
```

## DESCRIPTION

This function returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is issuing functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next function to be executed.

This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

Upon success, *t_look* returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

| | |
|---|---|
| T_LISTEN | connection indication received |
| T_CONNECT | connect confirmation received |
| T_DATA | normal data received |

| | |
|---|---|
| T_EXDATA | expedited data received |
| T_DISCONNECT | disconnect received |
| T_ERROR | fatal error indication |
| T_UDERR | datagram error indication |
| T_ORDREL | orderly release indication |

On failure, -1 is returned and *t_errno* is set to indicate the error.

## NAME

t_open – establish a transport endpoint

## SYNOPSIS

```
#include <tiuser.h>

int t_open(path, oflag, info)
char *path;
int oflag;
struct t_info *info;
```

## DESCRIPTION

*t_open* must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by opening an operating system file that identifies a particular transport provider (i.e. transport protocol) and returning a file descriptor that identifies that endpoint. For example, opening the file */dev/iso_cots* identifies an OSI connection-oriented transport layer protocol as the transport provider.

*Path* points to the path name of the file to open, and *oflag* identifies any open flags [as in *open*(2)]. *t_open* returns a file descriptor that will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the *t_info* structure. This argument points to a *t_info* which contains the following members:

```
long addr;          /* max size of the transport protocol address */
long options;       /* max number of bytes of protocol-specific options */
long tsdu;          /* max size of a transport service data unit (TSDU) */
long etsdu;         /* max size of an expedited transport service data unit (ETSDU) */
long connect;       /* max amount of data allowed on connection establishment functions */
long discon;        /* max amount of data allowed on t_snddis and t_rcvdis functions */
long servtype;      /* service type supported by the transport provider */
```

The values of the fields have the following meanings:

| | |
|---|---|
| *addr* | A value greater than or equal to zero indicates the maximum size of a transport protocol address; a value of -1 specifies that there is no limit on the address size; and a value of -2 specifies that the transport provider does not provide user access to transport protocol addresses. |

(3N)

| | |
|---|---|
| *options* | A value greater than or equal to zero indicates the maximum number of bytes of protocol-specific options supported by the provider; a value of -1 specifies that there is no limit on the option size; and a value of -2 specifies that the transport provider does not support user-settable options. |
| *tsdu* | A value greater than zero specifies the maximum size of a transport service data unit (TSDU); a value of zero specifies that the transport provider does not support the concept of TSDU, although it does support the sending of a data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of a TSDU; and a value of -2 specifies that the transfer of normal data is not supported by the transport provider. |
| *etsdu* | A value greater than zero specifies the maximum size of an expedited transport service data unit (ETSDU); a value of zero specifies that the transport provider does not support the concept of ETSDU, although it does support the sending of an expedited data stream with no logical boundaries preserved across a connection; a value of -1 specifies that there is no limit on the size of an ETSDU; and a value of -2 specifies that the transfer of expedited data is not supported by the transport provider. |
| *connect* | A value greater than or equal to zero specifies the maximum amount of data that may be associated with connection establishment functions; a value of -1 specifies that there is no limit on the amount of data sent during connection establishment; and a value of -2 specifies that the transport provider does not allow data to be sent with connection establishment functions. |
| *discon* | A value greater than or equal to zero specifies the maximum amount of data that may be associated with the *t_snddis* and *t_rcvdis* functions; a value of -1 specifies that there is no limit on the amount of data sent with these abortive release functions; and a value of -2 specifies that the transport provider does not allow data to be sent with the abortive release functions. |

| | |
|---|---|
| *servtype* | This field specifies the service type supported by the transport provider, as described below. |

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the *t_alloc* function may be used to allocate these buffers. An error will result if a transport user exceeds the allowed data size on any function.

The *servtype* field of *info* may specify one of the following values on return:

| | |
|---|---|
| T_COTS | The transport provider supports a connection-mode service but does not support the optional orderly release facility. |
| T_COTS_ORD | The transport provider supports a connection-mode service with the optional orderly release facility. |
| T_CLTS | The transport provider supports a connectionless-mode service. For this service type, *t_open* will return -2 for *etsdu*, *connect*, and *discon*. |

A single transport endpoint may support only one of the above services at one time.

If *info* is set to NULL by the transport user, no protocol information is returned by *t_open*.

On failure, *t_errno* may be set to the following:

| | |
|---|---|
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

open(2).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_open* returns a valid file descriptor on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_optmgmt – manage options for a transport endpoint

## SYNOPSIS

```
#include <tiuser.h>

int t_optmgmt(fd, req, ret)
int fd;
struct t_optmgmt *req;
struct t_optmgmt *ret;
```

## DESCRIPTION

The *t_optmgmt* function enables a transport user to retrieve, verify, or negotiate protocol options with the transport provider. *Fd* identifies a bound transport endpoint.

The *req* and *ret* arguments point to a *t_optmgmt* structure containing the following members:

```
struct netbuf opt;
long    flags;
```

The *opt* field identifies protocol options and the *flags* field is used to specify the action to take with those options.

The options are represented by a *netbuf* [see *intro*(3); also for *len, buf* and *maxlen*] structure in a manner similar to the address in *t_bind*. *Req* is used to request a specific action of the provider and to send options to the provider. *Len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. *Maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold. The actual structure and content of the options is imposed by the transport provider.

The *flags* field of *req* can specify one of the following actions:

| | |
|---|---|
| T_NEGOTIATE | This action enables the user to negotiate the values of the options specified in *req* with the transport provider. The provider will evaluate the requested options and negotiate the values, returning the negotiated values through *ret*. |
| T_CHECK | This action enables the user to verify whether the options specified in *req* are supported by the transport provider. On return, the *flags* field of *ret* will have either T_SUCCESS or T_FAILURE set to indicate to the user whether the options are supported. These flags are only meaningful for the T_CHECK request. |
| T_DEFAULT | This action enables a user to retrieve the default options supported by the transport provider into the *opt* field of *ret*. In *req*, the *len* field of *opt* must be zero and the *buf* field may be NULL. |

If issued as part of the connectionless-mode service, *t_optmgmt* may block due to flow control constraints. The function will not complete until the transport provider has processed all previously sent data units.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TOUTSTATE] | The function was issued in the wrong sequence. |
| [TACCES] | The user does not have permission to negotiate the specified options. |
| [TBADOPT] | The specified protocol options were in an incorrect format or contained illegal information. |
| [TBADFLAG] | An invalid flag was specified. |
| [TBUFOVFLW] | The number of bytes allowed for an incoming argument is not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded. |

| | |
|---|---|
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_getinfo(3N), t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_optmgmt* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_rcv – receive data or expedited data sent over a connection

## SYNOPSIS

```
int t_rcv(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int *flags;
```

## DESCRIPTION

This function receives either normal or expedited data. *Fd* identifies the local transport endpoint through which data will arrive, *buf* points to a receive buffer where user data will be placed, and *nbytes* specifies the size of the receive buffer. *Flags* may be set on return from *t_rcv* and specifies optional flags as described below.

By default, *t_rcv* operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_rcv* will execute in asynchronous mode and will fail if no data is available. (See TNODATA below.)

On return from the call, if T_MORE is set in *flags* this indicates that there is more data and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple *t_rcv* calls. Each *t_rcv* with the T_MORE flag set indicates that another *t_rcv* must follow immediately to get more data for the current TSDU. The end of the TSDU is identified by the return of a *t_rcv* call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open* or *t_getinfo*, the T_MORE flag is not meaningful and should be ignored.

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, *t_rcv* will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will not have T_EXPEDITED set on return. The end of the ETSDU is identified by the return of a *t_rcv* call with the T_MORE flag not set.

If expedited data arrives after part of a TSDU has been retrieved, receipt of the remainder of the TSDU will be suspended until the ETSDU has been processed. Only after the full ETSDU has been retrieved (T_MORE not set) will the remainder of the TSDU be available to the user.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TNODATA] | O_NDELAY was set, but no data is currently available from the transport provider. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N), t_snd(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

On successful completion, *t_rcv* returns the number of bytes received, and it returns -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_rcvconnect – receive the confirmation from a connect request

## SYNOPSIS

**#include <tiuser.h>**

**int t_rcvconnect(fd, call)**
**int fd;**
**struct t_call *call;**

## DESCRIPTION

This function enables a calling transport user to determine the status of a previously sent connect request and is used in conjunction with *t_connect* to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

*Fd* identifies the local transport endpoint where communication will be established, and *call* contains information associated with the newly established connection. *Call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*Netbuf* is described in *intro*(3). In *call, addr* returns the protocol address associated with the responding transport endpoint, *opt* presents any protocol-specific information associated with the connection, *udata* points to optional user data that may be returned by the destination transport user during connection establishment, and *sequence* has no meaning for this function.

The *maxlen* [see *netbuf* in *intro*(3)] field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be NULL, in which case no information is given to the user on return from *t_rcvconnect*. By default, *t_rcvconnect* executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr, opt,* and *udata* fields reflect values associated with the connection.

If O_NDELAY is set (via *t_open* or *fcntl*), *t_rcvconnect* executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, *t_rcvconnect* fails and returns immediately without waiting for the connection to be established. (See TNODATA below.)

*t_rcvconnect* must be re-issued at a later time to complete the connection establishment phase and retrieve the information returned in *call*.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for an incoming argument is not sufficient to store the value of that argument and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to DATAXFER. |
| [TNODATA] | O_NDELAY was set, but a connect confirmation has not yet arrived. |
| [TLOOK] | An asynchronous event has occurred on this transport connection and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_accept(3N), t_bind(3N), t_connect(3N), t_listen(3N), t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_rcvconnect* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_rcvdis – retrieve information from disconnect

## SYNOPSIS

**#include <tiuser.h>**

**t_rcvdis(fd, discon)**
**int fd;**
**struct t_discon *discon;**

## DESCRIPTION

This function is used to identify the cause of a disconnect, and to retrieve any user data sent with the disconnect. *Fd* identifies the local transport endpoint where the connection existed, and *discon* points to a *t_discon* structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

*Netbuf* is described in *intro*(3). *Reason* specifies the reason for the disconnect through a protocol-dependent reason code, *udata* identifies any user data that was sent with the disconnect, and *sequence* may identify an outstanding connect indication with which the disconnect is associated. *Sequence* is only meaningful when *t_rcvdis* is issued by a passive transport user who has executed one or more *t_listen* functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be NULL and any user data associated with the disconnect will be discarded. However, if a user has retrieved more than one outstanding connect indication (via *t_listen*) and *discon* is NULL, the user will be unable to identify with which connect indication the disconnect is associated.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |

| | |
|---|---|
| [TNODIS] | No disconnect indication currently exists on the specified transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for incoming data is not sufficient to store the data. The provider's state, as seen by the user, will change to T_IDLE, and the disconnect indication information to be returned in *discon* will be discarded. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_connect(3N), t_listen(3N), t_open(3N), t_snddis(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_rcvdis* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_rcvrel – acknowledge receipt of an orderly release indication

## SYNOPSIS

```
#include <tiuser.h>

t_rcvrel(fd)
int fd;
```

## DESCRIPTION

This function is used to acknowledge receipt of an orderly release indication. *Fd* identifies the local transport endpoint where the connection exists. After receipt of this indication, the user may not attempt to receive more data because such an attempt will block forever. However, the user may continue to send data over the connection if *t_sndrel* has not been issued by the user.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on *t_open* or *t_getinfo*.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TNOREL] | No orderly release indication currently exists on the specified transport endpoint. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N), t_sndrel(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_rcvrel* returns 0 on success and -1 on failure *t_errno* is set to indicate the error.

## NAME

t_rcvudata – receive a data unit

## SYNOPSIS

**#include <tiuser.h>**

```
int t_rcvudata(fd, unitdata, flags)
int fd;
struct t_unitdata *unitdata;
int *flags;
```

## DESCRIPTION

This function is used in connectionless mode to receive a data unit from another transport user. *Fd* identifies the local transport endpoint through which data will be received, *unitdata* holds information associated with the received data unit, and *flags* is set on return to indicate that the complete data unit was not received. *Unitdata* points to a *t_unitdata* structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The *maxlen* [see *netbuf*in *intro*(3)] field of *addr, opt,* and *udata* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, *addr* specifies the protocol address of the sending user, *opt* identifies protocol-specific options that were associated with this data unit, and *udata* specifies the user data that was received.

By default, *t_rcvudata* operates in synchronous mode and will wait for a data unit to arrive if none is currently available. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_rcvudata* will execute in asynchronous mode and will fail if no data units are available.

If the buffer defined in the *udata* field of *unitdata* is not large enough to hold the current data unit, the buffer will be filled and T_MORE will be set in *flags* on return to indicate that another *t_rcvudata* should be issued to retrieve the rest of the data unit. Subsequent *t_rcvudata* call(s) will return zero for the length of the address and options until the full data unit has been received.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TNODATA] | O_NDELAY was set, but no data units are currently available from the transport provider. |
| [TBUFOVFLW] | The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data information to be returned in *unitdata* will be discarded. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_rcvuderr(3N), t_sndudata(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_rcvudata* returns 0 on successful completion and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_rcvuderr – receive a unit data error indication

## SYNOPSIS

**#include <tiuser.h>**

**int t_rcvuderr(fd, uderr)**
**int fd;**
**struct t_uderr *uderr;**

## DESCRIPTION

This function is used in connectionless mode to receive information concerning an error on a previously sent data unit, and should only be issued following a unit data error indication. It informs the transport user that a data unit with a specific destination address and protocol options produced an error. *Fd* identifies the local transport endpoint through which the error report will be received, and *uderr* points to a *t_uderr* structure containing the following members: struct netbuf addr; struct netbuf opt; long error;

*Netbuf* is described in *intro*(3). The *maxlen* [see *netbuf* in *intro*(3)] field of *addr* and *opt* must be set before issuing this function to indicate the maximum size of the buffer for each.

On return from this call, the *addr* structure specifies the destination protocol address of the erroneous data unit, the *opt* structure identifies protocol-specific options that were associated with the data unit, and *error* specifies a protocol-dependent error code.

If the user does not care to identify the data unit that produced an error, *uderr* may be set to NULL and *t_rcvuderr* will simply clear the error indication without reporting any information to the user.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TNOUDERR] | No unit data error indication currently exists on the specified transport endpoint. |
| [TBUFOVFLW] | The number of bytes allocated for the incoming protocol address or options is not sufficient to store the information. The unit data error information to be returned in *uderr* will be discarded. |

| | |
|---|---|
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_rcvudata(3N), t_sndudata(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_rcvuderr* returns 0 on successful completion and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_snd – send data or expedited data over a connection

## SYNOPSIS

```
#include <tiuser.h>

int t_snd(fd, buf, nbytes, flags)
int fd;
char *buf;
unsigned nbytes;
int flags;
```

## DESCRIPTION

This function is used to send either normal or expedited data. *Fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below.

By default, *t_snd* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_snd* will execute in asynchronous mode, and will fail immediately if there are flow control restrictions.

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if O_NDELAY is set, it is possible that only part of the data will be accepted by the transport provider. In this case, *t_snd* will set T_MORE for the data that was sent (see below) and will return a value less than *nbytes*. If *nbytes* is zero, no data will be passed to the provider and *t_snd* will return zero.

If T_EXPEDITED is set in *flags*, the data will be sent as expedited data, and will be subject to the interpretations of the transport provider.

(3N)

If T_MORE is set in *flags*, or set as described above, an indication is sent to the transport provider that the transport service data unit (TSDU) (or expedited transport service data unit - ETSDU) is being sent through multiple *t_snd* calls. Each *t_snd* with the T_MORE flag set indicates that another *t_snd* will follow with more data for the current TSDU. The end of the TSDU (or ETSDU) is identified by a *t_snd* call with the T_MORE flag not set. Use of T_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the *info* argument on return from *t_open* or *t_getinfo*, the T_MORE flag is not meaningful and should be ignored.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as returned by *t_open* or *t_getinfo*. Failure to comply will result in protocol error EPROTO. (See TSYSERR below.)

If *t_snd* is issued from the T_IDLE state, the provider may silently discard the data. If *t_snd* is issued from any state other than T_DATAXFER or T_IDLE, the provider will generate an EPROTO error.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TFLOW] | O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N), t_rcv(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

On successful completion, *t_snd* returns the number of bytes accepted by the transport provider, and it returns -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_snddis – send user-initiated disconnect request

## SYNOPSIS

```
#include <tiuser.h>

int t_snddis(fd, call)
int fd;
struct t_call *call;
```

## DESCRIPTION

This function is used to initiate an abortive release on an already established connection or to reject a connect request. *Fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. *Call* points to a *t_call* structure which contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

*Netbuf* is described in *intro*(3). The values in *call* have different semantics, depending on the context of the call to *t_snddis*. When rejecting a connect request, *call* must be non-NULL and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* need only be used when data is being sent with the disconnect request. The *addr*, *opt*, and *sequence* fields of the *t_call* structure are ignored. If the user does not wish to send data to the remote user, the value of *call* may be NULL.

*Udata* specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider as returned by *t_open* or *t_getinfo*. If the *len* field of *udata* is zero, no data will be sent to the remote user.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |

| | |
|---|---|
| [TOUTSTATE] | The function was issued in the wrong sequence. The transport provider's outgoing queue may be flushed, so data may be lost. |
| [TBADDATA] | The amount of user data specified was not within the bounds allowed by the transport provider. The transport provider's outgoing queue will be flushed, so data may be lost. |
| [TBADSEQ] | An invalid sequence number was specified, or a NULL call structure was specified when rejecting a connect request. The transport provider's outgoing queue will be flushed, so data may be lost. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint and requires immediate attention. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_connect(3N), t_getinfo(3N), t_listen(3N), t_open(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_snddis* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_sndrel – initiate an orderly release

## SYNOPSIS

**#include <tiuser.h>**

**int t_sndrel(fd)**
**int fd;**

## DESCRIPTION

This function is used to initiate an orderly release of a transport connection and indicates to the transport provider that the transport user has no more data to send. *Fd* identifies the local transport endpoint where the connection exists. After issuing *t_sndrel,* the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has been received.

This function is an optional service of the transport provider, and is only supported if the transport provider returned service type T_COTS_ORD on *t_open* or *t_getinfo*.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TFLOW] | O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting the function at this time. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_open(3N), t_rcvrel(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_sndrel* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

## NAME

t_sndudata – send a data unit

## SYNOPSIS

```
#include <tiuser.h>

int t_sndudata(fd, unitdata)
int fd;
struct t_unitdata *unitdata;
```

## DESCRIPTION

This function is used in connectionless mode to send a data unit to another transport user. *Fd* identifies the local transport endpoint through which data will be sent, and *unitdata* points to a *t_unitdata* structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

*Netbuf* is described in *intro*(3). In *unitdata, addr* specifies the protocol address of the destination user, *opt* identifies protocol-specific options that the user wants associated with this request, and *udata* specifies the user data to be sent. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If the *len* field of *udata* is zero, no data unit will be passed to the transport provider; *t_sndudata* will not send zero-length data units.

By default, *t_sndudata* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NDELAY is set (via *t_open* or *fcntl*), *t_sndudata* will execute in asynchronous mode and will fail under such conditions.

If *t_sndudata* is issued from an invalid state, or if the amount of data specified in *udata* exceeds the TSDU size as returned by *t_open* or *t_getinfo*, the provider will generate an EPROTO protocol error. (See TSYSERR below.)

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TFLOW] | O_NDELAY was set, but the flow control mechanism prevented the transport provider from accepting data at this time. |
| [TNOTSUPPORT] | This function is not supported by the underlying transport provider. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

intro(3), t_rcvudata(3N), t_rcvuderr(3N).
*SYSTEM V/68 Programmer's Guide*.

## DIAGNOSTICS

*t_sndudata* returns 0 on successful completion and -1 on failure *t_errno* is set to indicate the error.

## NAME

t_sync – synchronize transport library

## SYNOPSIS

```
#include <tiuser.h>

int t_sync(fd)
int fd;
```

## DESCRIPTION

For the transport endpoint specified by *fd, t_sync* synchronizes the data structures managed by the transport library with information from the underlying transport provider. In doing so, it can convert a raw file descriptor [obtained via *open*(2), *dup*(2), or as a result of a *fork*(2) and *exec*(2)] to an initialized transport endpoint, assuming that file descriptor referenced a transport provider. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process *forks* a new process and issues an *exec*, the new process must issue a *t_sync* to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the provider. *t_sync* returns the current state of the provider to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes; it is possible that a process or an incoming event could change the provider's state *after* a *t_sync* is issued.

If the provider is undergoing a state transition when *t_sync* is called, the function will fail.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor is a valid open file descriptor but does not refer to a transport endpoint. |
| [TSTATECHNG] | The transport provider is undergoing a state change. |

| | |
|---|---|
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

dup(2), exec(2), fork(2), open(2).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_sync* returns the state of the transport provider on successful completion and -1 on failure and *t_errno* is set to indicate the error. The state returned may be one of the following:

| | |
|---|---|
| T_UNBND | unbound |
| T_IDLE | idle |
| T_OUTCON | outgoing connection pending |
| T_INCON | incoming connection pending |
| T_DATAXFER | data transfer |
| T_OUTREL | outgoing orderly release (waiting for an orderly release indication) |
| T_INREL | incoming orderly release (waiting for an orderly release request) |

## NAME

t_unbind – disable a transport endpoint

## SYNOPSIS

```
#include <tiuser.h>

int t_unbind(fd)
int fd;
```

## DESCRIPTION

The *t_unbind* function disables the transport endpoint specified by *fd* which was previously bound by *t_bind* (3N). On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider.

On failure, *t_errno* may be set to one of the following:

| | |
|---|---|
| [TBADF] | The specified file descriptor does not refer to a transport endpoint. |
| [TOUTSTATE] | The function was issued in the wrong sequence. |
| [TLOOK] | An asynchronous event has occurred on this transport endpoint. |
| [TSYSERR] | A system error has occurred during execution of this function. |

## SEE ALSO

t_bind(3N).
*SYSTEM V/68 Programmer's Guide*

## DIAGNOSTICS

*t_unbind* returns 0 on success and -1 on failure and *t_errno* is set to indicate the error.

(3N)

(3N)

## NAME

assert – verify program assertion

## SYNOPSIS

**#include <assert.h>**

**assert (expression)**
**int expression;**

## DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **-DNDEBUG** [see *cpp*(1)], or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include** <assert.h>" statement, will stop assertions from being compiled into the program.

## SEE ALSO

cpp(1), abort(3C).

## CAVEAT

Since *assert* is implemented as a macro, the *expression* may not contain any string literals.

(3X)

## NAME

crypt – password and file encryption functions

## SYNOPSIS

```
cc [flag ...] file ... –lcrypt

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

void setkey (key)
char *key;

void encrypt (block, flag)
char *block;
int flag;

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

void des_setkey (key)
char *key;

void des_encrypt (block, flag)
char *block;
int flag;

int run_setkey (p, key)
int p[2];
char *key;

int run_crypt (offset, buffer, count, p)
long offset;
char *buffer;
unsigned int count;
int p[2];

int crypt_close(p)
int p[2];
```

## NOTE

Decryption is not provided in the international version of *crypt*(3X). The international version is part of the C *Programming Language Utilities,* and the domestic version is part of the *Security Administration Utilities.* If decryption is attempted with the international version of *des_encrypt,* an error message is printed.

## DESCRIPTION

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

*Key* is a user's typed password. *Salt* is a two-character string chosen from the set [a-zA-Z0-9./]; 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 *des_setkey* and *des_encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *des_setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that will be used with the hashing algorithm to encrypt the string *block* with the function *des_encrypt*.

The argument to the *des_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 to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key set by *des_setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

*Crypt, setkey,* and *encrypt* are front-end routines that invoke *des_crypt, des_setkey,* and *des_encrypt* respectively.

The routines *run_setkey* and *run_crypt* are designed for use by applications that need cryptographic capabilities [such as *ed*(1) and *vi*(1)] that must be compatible with the *crypt*(1) user-level utility. *Run_setkey* establishes a two-way pipe connection with *crypt*(1), using *key* as the password argument. *Run_crypt* takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using *crypt*(1). *Offset* is the relative byte position from the beginning of the file that the block of text provided in *block* is coming from. *Count* is the number of characters in *block,* and *connection* is an array containing indices to a table of input and output file streams. When encryption is finished, *crypt_close* is used to terminate the connection with *crypt*(1).

*Run_setkey* returns -1 if a connection with *crypt*(1) cannot be established. This will occur on international versions of UNIX where *crypt*(1) is not available. If a null key is passed to *run_setkey,* 0 is returned. Otherwise, 1 is returned. *Run_crypt* returns -1 if it cannot write output or read input from the pipe attached to *crypt*. Otherwise it returns 0.

## DIAGNOSTICS

In the international version of *crypt*(3X), a flag argument of 1 to *des_encrypt* is not accepted, and an error message is printed.

## SEE ALSO

getpass(3C), passwd(4).
crypt(1), login(1), passwd(1) in the *User's Reference Manual.*

## CAVEAT

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

## NAME

curses – terminal screen handling and optimization package

## SYNOPSIS

The *curses* manual page is organized as follows:

In SYNOPSIS
- compiling information
- summary of parameters used by *curses* routines
- alphabetical list of curses routines, showing their parameters

In DESCRIPTION:
- An overview of how *curses* routines should be used

In ROUTINES, descriptions of each *curses* routines, are grouped under the appropriate topics:
- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of **curscr**

Then come sections on:
- ATTRIBUTES
- FUNCTION CALLS
- LINE GRAPHICS

cc [flag ...] file ... **–lcurses** [library ...]

**#include <curses.h>** (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the *curses* library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**.) Routines that return integers are not listed in the parameter list below.

bool bf

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp
char *cap, *capname, codename[2], erasechar, *filename, *fmt
char *keyname, killchar, *label, *longname
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ]
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ]
char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type

chtype attrs, ch, horch, vertch

FILE *infd, *outfd

int begin_x, begin_y, begline, bot, c, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fildes
int (*init( )), labfmt, labnum, line
int ms, ncols, new, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int p1, p2, p9, pmincol, pminrow, (*putc( )), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y

SCREEN *new, *newterm, *set_term

TERMINAL *cur_term, *nterm, *oterm

va_list varglist

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win

**addch(ch)**
**addstr(str)**
**attroff(attrs)**
**attron(attrs)**
**attrset(attrs)**

**baudrate**( )
**beep**( )
**box**(win, vertch, horch)
**cbreak**( )
**clear**( )
**clearok**(win, bf)
**clrtobot**( )
**clrtoeol**( )
**copywin**(srcwin, dstwin, sminrow, smincol, dminrow, dmincol,
    dmaxrow, dmaxcol, overlay)"
**curs_set**(visibility)
**def_prog_mode**( )
**def_shell_mode**( )
**del_curterm**(oterm)
**delay_output**(ms)
**delch**( )
**deleteln**( )
**delwin**(win)
**doupdate**( )
**draino**(ms)
**echo**( )
**echochar**(ch)
**endwin**( )
**erase**( )
**erasechar**( )
**filter**( )
**flash**( )
**flushinp**( )
**garbagedlines**(win, begline, numlines)
**getbegyx**(win, y, x)
**getch**( )
**getmaxyx**(win, y, x)
**getstr**(str)
**getsyx**(y, x)
**getyx**(win, y, x)
**halfdelay**(tenths)
**has_ic**( )
**has_il**( )
**idlok**(win, bf)
**inch**( )

**initscr**( )
**insch**(ch)
**insertln**( )
**intrflush**(win, bf)
**isendwin**( )
**keyname**(c)
**keypad**(win, bf)
**killchar**( )
**leaveok**(win, bf)
**longname**( )
**meta**(win, bf)
**move**(y, x)
**mvaddch**(y, x, ch)
**mvaddstr**(y, x, str)
**mvcur**(oldrow, oldcol, newrow, newcol)
**mvdelch**(y, x)
**mvgetch**(y, x)
**mvgetstr**(y, x, str)
**mvinch**(y, x)
**mvinsch**(y, x, ch)
**mvprintw**(y, x, fmt [, arg...])
**mvscanw**(y, x, fmt [, arg...])
**mvwaddch**(win, y, x, ch)
**mvwaddstr**(win, y, x, str)
**mvwdelch**(win, y, x)
**mvwgetch**(win, y, x)
**mvwgetstr**(win, y, x, str)
**mvwin**(win, y, x)
**mvwinch**(win, y, x)
**mvwinsch**(win, y, x, ch)
**mvwprintw**(win, y, x, fmt [, arg...])
**mvwscanw**(win, y, x, fmt [, arg...])
**napms**(ms)
**newpad**(nlines, ncols)
**newterm**(type, outfd, infd)
**newwin**(nlines, ncols, begin_y, begin_x)
**nl**( )
**nocbreak**( )
**nodelay**(win, bf)
**noecho**( )

**nonl**()
**noraw**()
**notimeout**(win, bf)
**overlay**(srcwin, dstwin)
**overwrite**(srcwin, dstwin)
**pechochar**(pad, ch)
**pnoutrefresh**(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
**prefresh**(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
**printw**(fmt [, arg...])
**putp**(str)
**raw**()
**refresh**()
**reset_prog_mode**()
**reset_shell_mode**()
**resetty**()
**restartterm**(term, fildes, errret)
**ripoffline**(line, init)
**savetty**()
**scanw**(fmt [, arg...])
**scr_dump**(filename)
**scr_init**(filename)
**scr_restore**(filename)
**scroll**(win)
**scrollok**(win, bf)
**set_curterm**(nterm)
**set_term**(new)
**setscrreg**(top, bot)
**setsyx**(y, x)
**setupterm**(term, fildes, errret)
**slk_clear**()
**slk_init**(fmt)
**slk_label**(labnum)
**slk_noutrefresh**()
**slk_refresh**()
**slk_restore**()
**slk_set**(labnum, label, fmt)
**slk_touch**()
**standend**()
**standout**()
**subpad**(orig, nlines, ncols, begin_y, begin_x)

**subwin**(orig, nlines, ncols, begin_y, begin_x)
**tgetent**(bp, name)
**tgetflag**(codename)
**tgetnum**(codename)
**tgetstr**(codename, area)
**tgoto**(cap, col, row)
**tigetflag**(capname)
**tigetnum**(capname)
**tigetstr**(capname)
**touchline**(win, start, count)
**touchwin**(win)
**tparm**(str, p1, p2, ..., p9)
**tputs**(str, count, putc)
**traceoff**()
**traceon**()
**typeahead**(fildes)
**unctrl**(c)
**ungetch**(c)
**vidattr**(attrs)
**vidputs**(attrs, putc)
**vwprintw**(win, fmt, varglist)
**vwscanw**(win, fmt, varglist)
**waddch**(win, ch)
**waddstr**(win, str)
**wattroff**(win, attrs)
**wattron**(win, attrs)
**wattrset**(win, attrs)
**wclear**(win)
**wclrtobot**(win)
**wclrtoeol**(win)
**wdelch**(win)
**wdeleteln**(win)
**wechochar**(win, ch)
**werase**(win)
**wgetch**(win)
**wgetstr**(win, str)
**winch**(win)
**winsch**(win, ch)
**winsertln**(win)
**wmove**(win, y, x)

**wnoutrefresh**(win)
**wprintw**(win, fmt [, arg...])
**wrefresh**(win)
**wscanw**(win, fmt [, arg...])
**wsetscrreg**(win, top, bot)
**wstandend**(win)
**wstandout**(win)

## DESCRIPTION

The *curses* routines give the user a terminal-independent method of updating screens with reasonable optimization.

In order to initialize the routines, the routine **initscr**() or **newterm**() must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine **endwin**() must be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented programs want this) after calling **initscr**() you should call "**cbreak**(); **noecho**();" Most programs would additionally call "**nonl**(); **intrflush** (**stdscr**, **FALSE**); **keypad**(**stdscr**, **TRUE**);".

Before a *curses* program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable TERM has been exported. For further details, see *profile*(4), *tput*(1), and the "Tabs and Initialization" subsection of *terminfo*(4).

The *curses* library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**(). Windows are referred to by variables declared as **WINDOW ***; the type **WINDOW** is defined in **<curses.h>** to be a C structure. These data structures are manipulated with routines described below, among which the most basic are **move**() and **addch**(). (More general versions of these routines are included with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**.) Then **refresh**() is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of **newpad()** under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE, ACS_HLINE,** and **KEY_LEFT.**

*curses* also defines the **WINDOW *** variable, **curscr,** which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok()** is **curscr,** the next call to **wrefresh()** with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh()** is **curscr,** the screen in immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override **terminfo**'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable **TERMINFO** is defined, any program using *curses* will check for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **att4425,** then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms, curses* will first check *$HOME/myterms/a/att4425*, and, if that fails, will then check */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in **<curses.h>**, and will be filled in by **initscr()** with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively. The constants **ERR** and **OK** are returned by routines to indicate whether the routine successfully completed. These constants are also defined in **<curses.h>**.

## ROUTINES

Many of the following routines have two or more versions. The routines prefixed with **w** require a *window* argument. The routines prefixed with p require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv()** routines imply a call to **move()** before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always (0,0), not (1,1). The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**win** and **pad** are always of type **WINDOW ***.) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE**. (*bf* is always of type **bool**.) The types **WINDOW**, **bool**, and **chtype** are defined in <curses.h>. See the SYNOPSIS for a summary of what types all variables are.

All routines return either the integer ERR or the integer OK, unless otherwise noted. Routines that return pointers always return NULL on error.

### Overall Screen Manipulation

WINDOW ***initscr()** The first routine called should almost always be **initscr()**. (The exceptions are **slk_init()**, **filter()**, and **ripoffline()**.) This will determine the terminal type and initialize all *curses* data structures. **initscr()** also arranges that the first call to **refresh()** will clear the screen. If errors occur, **initscr()** will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr** is returned. If the program wants an indication of error conditions, **newterm()** should be used instead of **initscr()**. **initscr()** should only be called once per application.

(3X)

**endwin()**
A program should always call **endwin()** before exiting or escaping from *curses* mode temporarily, to do a shell escape or *system*(3S) call, for example. This routine will restore *tty*(7) modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **wrefresh()** or **doupdate()**.

**isendwin()**
Returns **TRUE** if **endwin()** has been called without any subsequent calls to **wrefresh()**.

SCREEN ***newterm**(type, outfd, infd)
A program that outputs to more than one terminal must use **newterm()** for each terminal instead of **initscr()**. A program that wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm()** should be called once for each terminal. It returns a variable of type **SCREEN*** that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin()** for each terminal being used. If **newterm()** is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin()** is called.

SCREEN ***set_term**(new)
This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates **SCREEN** pointers; all other routines affect only the current terminal.

## Window and Pad Manipulation

**refresh()**
**wrefresh** (win)

These routines (or **prefresh()**, **pnoutrefresh()**, **wnoutrefresh()**, or **doupdate()**) must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh()** copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh()** does the same thing, except it uses **stdscr** as a default window. Unless **leaveok()** has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that **refresh()** is a macro.

**wnoutrefresh**(win)
**doupdate()**

These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh()** alone. How this is accomplished is described in the next paragraph.

*curses* keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh()** works by first calling **wnoutrefresh()**, which copys the named window to the virtual screen, and then by calling **doupdate()**, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to the screen. By first calling **wnoutrefresh()** for each window, it is then possible to call **doupdate()** once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW ***newwin**(nlines, ncols, begin_y, begin_x)

Create and return a pointer to a new window with the given number of lines (or rows), *nlines,* and columns, *ncols.* The upper left corner of the window is at line *begin_y,* column *begin_x.* If either *nlines* or *ncols* is **0**, they will be set to the value of **lines**–*begin_y* and **cols**–*begin_x.* A new full-screen window is created by calling **newwin(0,0,0,0)**.

**mvwin**(win, y, x)

Move the window so that the upper left corner will be at position (*y, x*). If the move would cause the window to be off the screen, it is an error and the window is not moved.

WINDOW ***subwin**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a new window with the given number of lines (or rows), *nlines,* and columns, *ncols.* The window is at position (*begin_y, begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig.*) The window is made in the middle of the window *orig,* so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **wrefresh**( ).

**delwin**(win)

Delete the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

WINDOW ***newpad**(nlines, ncols)

Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines,* and columns, *ncols.* A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call **wrefresh**( ) with a pad as an argument; the routines **prefresh**( ) or **pnoutrefresh**( ) should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW ***subpad**(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin()**, which uses screen coordinates, the window is at position (*begin_y, begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin()** or **touchline()** on *orig* before calling **prefresh()**.

**prefresh**(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
**pnoutrefresh**(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

These routines are analogous to **wrefresh()** and **wnoutrefresh()** except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow, smincol, smaxrow,* and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow, pmincol, sminrow,* or *smincol* are treated as if they were zero.

## Output

These routines are used to "draw" text on windows.

**addch**(ch)
**waddch**(win, ch)
**mvaddch**(y, x, ch)
**mvwaddch**(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar* (see *putc*(3S)). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok()** is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **clrtoeol()** before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the ^X notation. (Calling **winch()** after adding a control character will not return the control character, but instead will return the representation of the control character.)

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **inch()** and **addch()**.) See **standout()**, below.

Note that *ch* is actually of type **chtype**, not a character.

Note that **addch()**, **mvaddch()**, and **mvwaddch()**, are macros.

**echochar**(ch)
**wechochar**(win, ch)
**pechochar**(pad, ch) These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh**(), a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**(), the last location of the pad on the screen is reused for the arguments to **prefresh**().

Note that *ch* is actually of type **chtype**, not a character.

Note that **echochar**() is a macro.

**addstr**(str)
**waddstr**(win, str)
**mvwaddstr**(win, y, x, str)
**mvaddstr**(y, x, str) These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch**() once for each character in the string.

Note that **addstr**(), **mvaddstr**(), and **mvwaddstr**() are macros.

**attroff**(attrs)
**wattroff**(win, attrs)
**attron**(attrs)
**wattron**(win, attrs)
**attrset**(attrs)
**wattrset**(win, attrs)
**standend**()
**wstandend**(win)

**standout()**
**wstandout(win)**

These routines manipulate the current attributes of the named window. These attributes can be any combination of A_STANDOUT, A_REVERSE, A_BOLD, A_DIM, A_BLINK, A_UNDERLINE, and A_ALTCHARSET. These constants are defined in <**curses.h**> and can be combined with the C logical OR ( | ) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch()**. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of the characters put on the screen.

**attrset**(attrs) sets the current attributes of the given window to *attrs*. **attroff**(attrs) turns off the named attributes without turning on or off any other attributes. **attron**(attrs) turns on the named attributes without affecting any others. **standout()** is the same as **attron(A_STANDOUT)**. **standend()** is the same as **attrset (0)**, that is, it turns off all attributes.

Note that *attrs* is actually of type **chtype**, not a character.

Note that **attroff()**, **attron()**, **attrset()**, **standend()**, and **standout()** are macros.

**beep()**
**flash()**

These routines are used to signal the terminal user. **beep()** will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash()** will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

**box(win, vertch, horch)**
A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note that *vertch* and *horch* are actually of type **chtype**, not characters.

**erase()**
**werase(win)**
These routines copy blanks to every position in the window.

Note that **erase()** is a macro.

**clear()**
**wclear(win)**
These routines are like **erase()** and **werase()**, but they also call **clearok()**, arranging that the screen will be cleared completely on the next call to **wrefresh()** for that window, and repainted from scratch.

Note that **clear()** is a macro.

**clrtobot()**
**wclrtobot(win)**
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot()** is a macro.

**clrtoeol()**
**wclrtoeol(win)**
The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol()** is a macro.

**delay_output(ms)**
Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

| | |
|---|---|
| delch()<br>wdelch(win)<br>mvdelch(y, x)<br>mvwdelch(win, y, x) | The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "delete-character" feature.)<br><br>Note that **delch()**, **mvdelch()**, and **mvwdelch()** are macros. |
| deleteln()<br>wdeleteln(win) | The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware "delete-line" feature.)<br><br>Note that **deleteln()** is a macro. |
| getyx(win, y, x) | The cursor position of the window is placed in the two integer variables *y* and *x*. This is implemented as a macro, so no **"&"** is necessary before the variables. |
| getbegyx(win, y, x)<br>getmaxyx(win, y, x) | Like **getyx()**, these routines store the current beginning coordinates and size of the specified window.<br><br>Note that **getbegyx()** and **getmaxyx()** are macros. |
| insch(ch)<br>winsch(win, ch)<br>mvwinsch(win, y, x, ch)<br>mvinsch(y, x, ch) | The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.) |

(3X)

Note that *ch* is actually of type **chtype**, not a character.

Note that **insch()**, **mvinsch()**, and **mvwinsch()** are macros.

**insertln()**
**winsertln**(win)
A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that **insertln()** is a macro.

**move**(y, x)
**wmove**(win, y, x)
The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **refresh()** is called. The position specified is relative to the upper left corner of the window, which is (0, 0).

Note that **move()** is a macro.

**overlay**(srcwin, dstwin)
**overwrite**(srcwin, dstwin)
These routines overlay *srcwin* on top of *dstwin*; that is, all text in *srcwin* is copied into *dstwin*. *scrwin* and *dstwin* need not be the same size; only text where the two windows overlap is copied. The difference is that **overlay()** is non-destructive (blanks are not copied), while **overwrite()** is destructive.

**copywin**(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
This routine provides a finer grain of control over the **overlay()** and **overwrite()** routines. Like in the **prefresh()** routine, a rectangle is specified in the destination window, (*dminrow, dmincol*) and (*dmaxrow, dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow, smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay()**.

(3X)

**printw**(fmt [, arg...])
**wprintw**(win, fmt [, arg...])
**mvprintw**(y, x, fmt [, arg...])
**mvwprintw**(win, y, x, fmt [, arg...])

These routines are analogous to **printf**(3). The string which would be output by **printf**(3) is instead output using **waddstr**() on the given window.

**vwprintw**(win, fmt, varglist)

This routine corresponds to *vfprintf*(3S). It performs a **wprintw**() using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in **<varargs.h>**. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

**scroll**(win)

The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

**touchwin**(win)
**touchline**(win, start, count)

Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**() only pretends that *count* lines have been changed, beginning with line *start* .

**Input**

**getch**()
**wgetch**(win)
**mvgetch**(y, x)
**mvwgetch**(win, y, x) A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value **ERR** is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window. No **refresh**( ) will occur between the **move**( ) and the **getch**( ) done within the routines **mvgetch**( ) and **mvwgetch**( ).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**( )) and ECHO mode (**echo**( )) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If **keypad**(win, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. (See **keypad**( ) under "Input Options Setting.") Possible function keys are defined in **<curses.h>** with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of

(3X)

the escape key for a single character routine is discouraged. Also see **notimeout()** below.)

Note that **getch()**, **mvgetch()**, and **mvwgetch()** are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **getch()** is made, until a newline, carriage return, or enter key is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. As in **mvgetch()**, no **refresh()** is done between the **move()** and **getstr()** within the routines **mvgetstr()** and **mvwgetstr()**.

Note that **getstr()**, **mvgetstr()**, and **mvwgetstr()** are macros.

flushinp()
Throws away any typeahead that has been typed by the user and has not yet been read by the program.

ungetch(c)
Place *c* back onto the input queue to be returned by the next call to **wgetch()**.

inch()
winch(win)
mvinch(y, x)
mvwinch(win, y, x)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants A_CHARTEXT and A_ATTRIBUTES, defined in **<curses.h>**, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that **inch()**, **winch()**, **mvinch()**, and **mvwinch()** are macros.

**(3X)**

**scanw**(fmt [, arg...])
**wscanw**(win, fmt [, arg...])
**mvscanw**(y, x, fmt [, arg...])
**mvwscanw**(win, y, x, fmt [, arg...])

These routines correspond to *scanf*(3S), as do their arguments and return values. **wgetstr**() is called on the window, and the resulting line is used as input for the scan.

**vwscanw**(win, fmt, ap)

This routine is similar to **vwprintw**() above in that performs a **wscanw**() using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <varargs.h>. See the *vprintf*(3S) and *varargs*(5) manual pages for a detailed description on how to use variable argument lists.

## Output Options Setting

These routines set options within *curses* that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**().

**clearok**(win, bf) If enabled (*bf* is TRUE), the next call to **wrefresh**() with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

**idlok**(win, bf) If enabled (*bf* is TRUE), *curses* will consider using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is FALSE), *curses* will very seldom use this feature. (The "insert/delete-character" feature is always considered.) This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it isn't really needed. If "insert/delete-line" cannot be used, *curses* will redraw the changed portions of all lines.

**leaveok**(win, bf) Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

**setscrreg**(top, bot)
**wsetscrreg**(win, top, bot)

These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok**( ) are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok**( ) is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.)

Note that **setscrreg**( ) and **wsetscrreg**( ) are macros.

**scrollok**(win, bf) This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is **FALSE**), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh**( ) is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**( ).)

**nl()**
**nonl()**

These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using **nonl()**, *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

### Input Options Setting

These routines set options within *curses* that deal with input. The options involve using *ioctl*(2) and therefore interact with *curses* routines. It is not necessary to turn these options off before calling **endwin()**.

For more information on these options, see Chapter 10 of the *Programmer's Guide*.

**cbreak()**
**nocbreak()**

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see *termio*(7)). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak()** or **nocbreak()** explicitly. Most interactive programs using *curses* will set CBREAK mode.

Note that **cbreak()** overrides **raw()**. See **getch()** under "Input" for a discussion of how these routines interact with **echo()** and **noecho()**.

**echo()**
**noecho()**

These routines control whether characters typed by the user are echoed by **getch()** as they are typed. Echoing by the tty driver is always disabled, but initially **getch()** is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho()**. See **getch()** under "Input" for a discussion of how these routines interact with **cbreak()** and **nocbreak()**.

**halfdelay**(tenths) Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed. *tenths* must be a number between 1 and 255. Use **nocbreak()** to leave half-delay mode.

**intrflush**(win, bf) If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

**keypad**(win, bf) This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch()** will return a single value representing the function key, as in **KEY_LEFT**. If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when **wgetch()** is called.

**meta**(win, bf) If enabled, characters returned by **wgetch()** are transmitted with all 8 bits, instead of with the highest bit stripped. In order for **meta()** to work correctly, the **km** (has_meta_key) capability has to be specified in the terminal's **terminfo**(4) entry.

**nodelay**(win, bf) This option causes **wgetch**( ) to be a non-blocking call. If no input is ready, **wgetch**( ) will return ERR. If disabled, **wgetch**( ) will hang until a key is pressed.

**notimeout**(win, bf) While interpreting an input escape sequence, **wgetch**( ) will set a timer while waiting for the next character. If **notimeout**(win, TRUE) is called, then **wgetch**( ) will not set a timer. The purpose of the timeout is to differentiate between sequences received from a function key and those typed by a user.

**raw**( )
**noraw**( ) The terminal is placed into or out of raw mode. RAW mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal. RAW mode also causes 8-bit input and output. The behavior of the BREAK key depends on other bits in the *tty*(7) driver that are not set by *curses*.

**typeahead**(fildes) *curses* does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **refresh**( ) or **doupdate**( ) is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm**( ), or **stdin** in the case that **initscr**( ) was used, will be used to do this typeahead checking. The **typeahead**( ) routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **–1**, then no typeahead checking will be done.

Note that *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

## Environment Queries

| | |
|---|---|
| baudrate( ) | Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer. |
| char erasechar( ) | The user's current erase character is returned. |
| has_ic( ) | True if the terminal has insert- and delete-character capabilities. |
| has_il( ) | True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok( )**. |
| char killchar( ) | The user's current line-kill character is returned. |
| char *longname( ) | This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr( )** or **newterm( )**. The area is overwritten by each call to **newterm( )** and is not restored by **set_term( )**, so the value should be saved between calls to **newterm( )** if **longname( )** is going to be used with multiple terminals. |

## Soft Labels

If desired, *curses* will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, *curses* will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. *curses* standardizes on 8 labels of 8 characters each.

| | |
|---|---|
| slk_init(labfmt) | In order to use soft labels, this routine must be called before **initscr( )** or **newterm( )** is called. If **initscr( )** winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to 0 indicates that the labels are to be arranged in a 3-2-3 arrangement; 1 asks for a 4-4 arrangement. |

slk_set(labnum, label, labfmt)
: *labnum* is the label number, from 1 to 8. *label* is the string to be put on the label, up to 8 characters in length. A **NULL** string or a **NULL** pointer will put up a blank label. *labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

**slk_refresh()**
**slk_noutrefresh()** These routines correspond to the routines **wrefresh()** and **wnoutrefresh()**. Most applications would use **slk_noutrefresh()** because a **wrefresh()** will most likely soon follow.

char *slk_label(labnum)
: The current label for label number *labnum*, with leading and trailing blanks stripped, is returned.

slk_clear() The soft labels are cleared from the screen.

slk_restore() The soft labels are restored to the screen after a **slk_clear()**.

slk_touch() All of the soft labels are forced to be output the next time a **slk_noutrefresh()** is performed.

Low-Level *curses* Access

The following routines give low-level access to various *curses* functionality. These routines typically would be used inside of library routines.

**def_prog_mode()**
**def_shell_mode()** Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode()** and **reset_shell_mode()** routines. This is done automatically by **initscr()**.

reset_prog_mode()
reset_shell_mode() Restore the terminal to "program" (in **curses**) or "shell" (out of *curses*) state. These are done automatically by **endwin()** and **doupdate()** after an **endwin()**, so they normally would not be called.

(3X)

**resetty()**
**savetty()**

These routines save and restore the state of the terminal modes. **savetty()** saves the current state of the terminal in a buffer and **resetty()** restores the state to what it was at the last call to **savetty()**.

getsyx(y, x)

The current coordinates of the virtual screen cursor are returned in *y* and *x*. Like **getyx()**, the variables *y* and *x* do not take an "&" before them. If **leaveok()** is currently **TRUE**, then **–1,–1** will be returned. If lines may have been removed from the top of the screen using **ripoffline()** and the values are to be used beyond just passing them on to **setsyx()**, the value *y*+**stdscr–>_yoffset** should be used for those other uses.

Note that **getsyx()** is a macro.

setsyx(y, x)

The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **–1**, then **leaveok()** will be set. The two routines **getsyx()** and **setsyx()** are designed to be used by a library routine which manipulates curses windows but does not want to mess up the current position of the program's cursor. The library routine would call **getsyx()** at the beginning, do its manipulation of its own windows, do a **wnoutrefresh()** on its windows, call **setsyx()**, and then call **doupdate()**.

**ripoffline**(line, init)

This routine provides access to the same facility that **slk_init()** uses to reduce the size of the screen. **ripoffline()** must be called before **initscr()** or **newterm()** is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr()**, the routine *init*() is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and

COLS (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh**( ) or **doupdate**( ) must not be called. It is allowable to call **wnoutrefresh**( ) during the initialization routine.

**ripoffline**( ) can be called up to five times before calling **initscr**( ) or **newterm**( ).

**scr_dump**(filename) The current contents of the virtual screen are written to the file *filename*.

**scr_restore**(filename) The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**(). The next call to **doupdate**( ) will restore the screen to what it looked like in the dump file.

**scr_init**(filename) The contents of *filename* are read in and used to initialize the *curses* data structures about what the terminal currently has on its screen. If the data is determined to be valid, *curses* will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**( ) would be used after **initscr**( ) or a *system*(3S) call to share the screen with another process which has done a **scr_dump**() after its **endwin**( ) call. The data will be declared invalid if the time-stamp of the tty is old or the *terminfo*(4) capability **nrrmc** is true.

**curs_set**(visibility) The cursor is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**.

**draino**(ms) Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

**garbagedlines**(win, begline, numlines)
This routine indicates to *curses* that a screen line is garbaged and should be thrown away before having anything written over the top of it. It could be used for programs such as editors which want a command to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire screen would be subject to even more communication noise. Just redrawing the single line gives some

semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

**napms**(ms) Sleep for *ms* milliseconds.

## Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo*(4) database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, **setupterm**( ) should be called. (Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).) This will define the set of terminal-dependent variables defined in the *terminfo*(4) database. The *terminfo*(4) variables **lines** and **columns** (see *terminfo*(4)) are initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist and the program is running in a layer (see *layers*(1)), the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo*(4) database are used.

The header files **<curses.h>** and **<term.h>** should be included, in this order, to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm**( ) to instantiate them. All *terminfo*(4) strings (including the output of **tparm**( )) should be printed with **tputs**( ) or **putp**( ). Before exiting, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo*(4)). (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the *curses* routines (see **endwin**( )).

**setupterm**(term, fildes, errret)

Reads in the *terminfo*(4) database, initializing the *terminfo*(4) structures, but does not set up the output virtualization structures used by *curses*. The terminal type is in the character string *term*; if *term* is NULL, the environment variable TERM will be used. All output is to the file descriptor *fildes*. If *errret* is not NULL, then **setupterm**( ) will return OK or ERR and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **-1** means that the *terminfo*(4) database could not be found. If *errret* is NULL, **setupterm()** will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The *terminfo*(4) boolean, numeric and string variables are stored in a structure of type TERMINAL. After **setupterm()** returns successfully, the variable **cur_term** (of type TERMINAL *) is initialized with all of the information that the *terminfo*(4) boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm()** again. Further calls to **setupterm()** will allocate new space rather than reuse the space pointed to by **cur_term**.

**set_curterm**(nterm) *nterm* is of type TERMINAL *. **set_curterm()** sets the variable **cur_term** to *nterm*, and makes all of the *terminfo*(4) boolean, numeric and string variables use the values from *nterm*.

**del_curterm**(oterm) *oterm* is of type TERMINAL *. **del_curterm()** frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo*(4) boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm()** has been called.

**restartterm**(term, fildes, errret)
Like **setupterm()** after a memory restore.

**char *tparm**(str, $p_1$, $p_2$, . . ., $p_9$)
Instantiate the string *str* with parms $p_i$. A pointer is returned to the result of *str* with the parameters applied.

(3X)

**tputs**(str, count, putc)
: Apply padding to the string *str* and output it. *str* must be a *terminfo*(4) string variable or the return value from **tparm**(), **tgetstr**(), **tigetstr**() or **tgoto**(). *count* is the number of lines affected, or **1** if not applicable. *putc*() is a *putchar*(3S)-like routine to which the characters are passed, one at a time.

**putp**(str)
: A routine that calls **tputs** (*str*, **1**, **putchar**()).

**vidputs**(attrs, putc)
: Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*(3S)-like routine *putc*().

**vidattr**(attrs)
: Like **vidputs**(), except that it outputs through *putchar*(3S).

**mvcur**(oldrow, oldcol, newrow, newcol)
: Low-level cursor motion.

The following routines return the value of the capability corresponding to the *terminfo*(4) *capname* passed to them, such as **xenl**.

**tigetflag**(capname)
: The value **–1** is returned if *capname* is not a boolean capability.

**tigetnum**(capname)
: The value **–2** is returned if *capname* is not a numeric capability.

**tigetstr**(capname)
: The value (char *) **–1** is returned if *capname* is not a string capability.

char ***boolnames**[ ], ***boolcodes**[ ], ***boolfnames**[ ]
char ***numnames**[ ], ***numcodes**[ ], ***numfnames**[ ]
char ***strnames**[ ], ***strcodes**[ ], ***strfnames**[ ]
: These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo*(4) variables.

## Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo*(4) database.

| | |
|---|---|
| tgetent(bp, name) | Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*. |
| tgetflag(codename) | Get the boolean entry for *codename*. |
| tgetnum(codes) | Get numeric entry for *codename*. |
| char *tgetstr(codename, area) | Return the string entry for *codename*. If *area* is not **NULL**, then also store it in the buffer pointed to by *area* and advance *area*. **tputs**( ) should be used to output the returned string. |
| char *tgoto(cap, col, row) | Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**( ). |
| tputs(str, affcnt, putc) | See **tputs**( ) above, under "Terminfo-Level Manipulations". |

Miscellaneous

| | |
|---|---|
| **traceoff**( )<br>**traceon**( ) | Turn off and on debugging trace output when using the debug version of the *curses* library, */usr/lib/libdcurses.a*. This facility is available only to customers with a source license. |
| unctrl(c) | This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.<br><br>**unctrl**( ) is a macro, defined in <**unctrl.h**>, which is automatically included by <**curses.h**>. |
| char *keyname(c) | A character string corresponding to the key *c* is returned. |
| filter( ) | This routine is one of the few that is to be called before **initscr**( ) or **newterm**( ) is called. It arranges things so that *curses* thinks that there is a 1-line screen. *curses* will not use any terminal capabilities that assume that they know what line on the screen the cursor is on. |

### Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. (This is how most programs would implement a "repaint-screen" routine.) The source window argument to **overlay**( ), **overwrite**( ), and **copywin**( ) may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

### Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

| | |
|---|---|
| **crmode**( ) | Replaced by **cbreak**( ). |
| **fixterm**( ) | Replaced by **reset_prog_mode**( ). |
| **gettmode**( ) | A no-op. |
| **nocrmode**( ) | Replaced by **nocbreak**( ). |
| **resetterm**( ) | Replaced by **reset_shell_mode**( ). |
| **saveterm**( ) | Replaced by **def_prog_mode**( ). |
| **setterm**( ) | Replaced by **setupterm**( ). |

## ATTRIBUTES

The following video attributes, defined in <**curses.h**>, can be passed to the routines **attron**( ), **attroff**( ), and **attrset**( ), or OR'ed with the characters passed to **addch**( ).

| | |
|---|---|
| A_STANDOUT | Terminal's best highlighting mode |
| A_UNDERLINE | Underlining |
| A_REVERSE | Reverse video |
| A_BLINK | Blinking |
| A_DIM | Half bright |
| A_BOLD | Extra bright or bold |
| A_ALTCHARSET | Alternate character set |
| A_CHARTEXT | Bit-mask to extract character (described under **winch**( )) |
| A_ATTRIBUTES | Bit-mask to extract attributes (described under **winch**( )) |
| A_NORMAL | Bit mask to reset all attributes off (for example: **attrset (A_NORMAL)** |

FUNCTION-KEYS

The following function keys, defined in <**curses.h**>, might be returned by **getch**() if **keypad**() has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo*(4) database.

| *Name* | *Value* | *Key name* |
|---|---|---|
| KEY_BREAK | 0401 | break key (unreliable) |
| KEY_DOWN | 0402 | The four arrow keys ... |
| KEY_UP | 0403 | |
| KEY_LEFT | 0404 | |
| KEY_RIGHT | 0405 | ... |
| KEY_HOME | 0406 | Home key (upward+left arrow) |
| KEY_BACKSPACE | 0407 | backspace (unreliable) |
| KEY_F0 | 0410 | Function keys. Space for 64 keys is reserved. |
| KEY_F(n) | (KEY_F0+(n)) | Formula for $f_n$. |
| KEY_DL | 0510 | Delete line |
| KEY_IL | 0511 | Insert line |
| KEY_DC | 0512 | Delete character |
| KEY_IC | 0513 | Insert char or enter insert mode |
| KEY_EIC | 0514 | Exit insert char mode |
| KEY_CLEAR | 0515 | Clear screen |
| KEY_EOS | 0516 | Clear to end of screen |
| KEY_EOL | 0517 | Clear to end of line |
| KEY_SF | 0520 | Scroll 1 line forward |
| KEY_SR | 0521 | Scroll 1 line backwards (reverse) |
| KEY_NPAGE | 0522 | Next page |
| KEY_PPAGE | 0523 | Previous page |
| KEY_STAB | 0524 | Set tab |
| KEY_CTAB | 0525 | Clear tab |
| KEY_CATAB | 0526 | Clear all tabs |
| KEY_ENTER | 0527 | Enter or send |
| KEY_SRESET | 0530 | soft (partial) reset |
| KEY_RESET | 0531 | reset or hard reset |
| KEY_PRINT | 0532 | print or copy |

| | | |
|---|---|---|
| KEY_LL | 0533 | home down or bottom (lower left) keypad is arranged like this:<br>A1 up A3<br>left B2 right<br>C1 down C3 |
| KEY_A1 | 0534 | Upper left of keypad |
| KEY_A3 | 0535 | Upper right of keypad |
| KEY_B2 | 0536 | Center of keypad |
| KEY_C1 | 0537 | Lower left of keypad |
| KEY_C3 | 0540 | Lower right of keypad |
| KEY_BTAB | 0541 | Back tab key |
| KEY_BEG | 0542 | beg(inning) key |
| KEY_CANCEL | 0543 | cancel key |
| KEY_CLOSE | 0544 | close key |
| KEY_COMMAND | 0545 | cmd (command) key |
| KEY_COPY | 0546 | copy key |
| KEY_CREATE | 0547 | create key |
| KEY_END | 0550 | end key |
| KEY_EXIT | 0551 | exit key |
| KEY_FIND | 0552 | find key |
| KEY_HELP | 0553 | help key |
| KEY_MARK | 0554 | mark key |
| KEY_MESSAGE | 0555 | message key |
| KEY_MOVE | 0556 | move key |
| KEY_NEXT | 0557 | next object key |
| KEY_OPEN | 0560 | open key |
| KEY_OPTIONS | 0561 | options key |
| KEY_PREVIOUS | 0562 | previous object key |
| KEY_REDO | 0563 | redo key |
| KEY_REFERENCE | 0564 | ref(erence) key |
| KEY_REFRESH | 0565 | refresh key |
| KEY_REPLACE | 0566 | replace key |
| KEY_RESTART | 0567 | restart key |
| KEY_RESUME | 0570 | resume key |
| KEY_SAVE | 0571 | save key |
| KEY_SBEG | 0572 | shifted beginning key |
| KEY_SCANCEL | 0573 | shifted cancel key |
| KEY_SCOMMAND | 0574 | shifted command key |
| KEY_SCOPY | 0575 | shifted copy key |
| KEY_SCREATE | 0576 | shifted create key |

| | | |
|---|---|---|
| KEY_SDC | 0577 | shifted delete char key |
| KEY_SDL | 0600 | shifted delete line key |
| KEY_SELECT | 0601 | select key |
| KEY_SEND | 0602 | shifted end key |
| KEY_SEOL | 0603 | shifted clear line key |
| KEY_SEXIT | 0604 | shifted exit key |
| KEY_SFIND | 0605 | shifted find key |
| KEY_SHELP | 0606 | shifted help key |
| KEY_SHOME | 0607 | shifted home key |
| KEY_SIC | 0610 | shifted input key |
| KEY_SLEFT | 0611 | shifted left arrow key |
| KEY_SMESSAGE | 0612 | shifted message key |
| KEY_SMOVE | 0613 | shifted move key |
| KEY_SNEXT | 0614 | shifted next key |
| KEY_SOPTIONS | 0615 | shifted options key |
| KEY_SPREVIOUS | 0616 | shifted prev key |
| KEY_SPRINT | 0617 | shifted print key |
| KEY_SREDO | 0620 | shifted redo key |
| KEY_SREPLACE | 0621 | shifted replace key |
| KEY_SRIGHT | 0622 | shifted right arrow |
| KEY_SRSUME | 0623 | shifted resume key |
| KEY_SSAVE | 0624 | shifted save key |
| KEY_SSUSPEND | 0625 | shifted suspend key |
| KEY_SUNDO | 0626 | shifted undo key |
| KEY_SUSPEND | 0627 | suspend key |
| KEY_UNDO | 0630 | undo key |

## LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch()**. When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default charcter listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

| *Name* | *Default* | *Glyph Description* |
|---|---|---|
| ACS_ULCORNER | + | upper left corner |
| ACS_LLCORNER | + | lower left corner |
| ACS_URCORNER | + | upper right corner |
| ACS_LRCORNER | + | lower right corner |
| ACS_RTEE | + | right tee (┤) |
| ACS_LTEE | + | left tee (├) |

| | | |
|---|---|---|
| ACS_BTEE | + | bottom tee (⊥) |
| ACS_TTEE | + | top tee (T) |
| ACS_HLINE | – | horizontal line |
| ACS_VLINE | \| | vertical line |
| ACS_PLUS | + | plus |
| ACS_S1 | – | scan line 1 |
| ACS_S9 | _ | scan line 9 |
| ACS_DIAMOND | + | diamond |
| ACS_CKBOARD | : | checker board (stipple) |
| ACS_DEGREE | ' | degree symbol |
| ACS_PLMINUS | # | plus/minus |
| ACS_BULLET | o | bullet |
| ACS_LARROW | < | arrow pointing left |
| ACS_RARROW | > | arrow pointing right |
| ACS_DARROW | v | arrow pointing down |
| ACS_UARROW | ^ | arrow pointing up |
| ACS_BOARD | # | board of squares |
| ACS_LANTERN | # | lantern symbol |
| ACS_BLOCK | # | solid square block |

## RETURN VALUES

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **setscrreg()**, **wsetscrreg()**, **getsyx()**, **getyx()**, **getbegy()**, **getmaxyx()**. For these macros, no useful value is returned.

Routines that return pointers always return **(type *) NULL** on error.

## BUGS

Currently typeahead checking is done using a nodelay read followed by an **ungetch()** of any character that may have been read. Typeahead checking is done only if **wgetch()** has been called at least once. This will be changed when proper kernel support is available. Programs which use a mixture of their own input routines with *curses* input routines may wish to call **typeahead(–1)** to turn off typeahead checking.

The argument to **napms()** is currently rounded up to the nearest second.

**draino** (ms) only works for *ms* equal to **0**.

## WARNINGS

To use the new *curses* features, use the Release 3 version of *curses* on

Release 3 of the operating system. All programs that ran with Release 2 *curses* will run with Release 3. You may link applications with object files based on the Release 2 *curses/terminfo* with the Release 3.0 *libcurses.a* library. You may link applications with object files based on the Release 3.0 *curses/terminfo* with the Release 2 *libcurses.a* library, so long as the application does not use the new features in the Release 3.0 *curses/terminfo*.

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names **erase**( ) and **move**( ). The *curses* versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or **#undef move**( ) and **erase**( ) in the *plot*(3X) code.

Between the time a call to **initscr**( ) and **endwin**( ) has been issued, use only the routines in the *curses* library to generate output. Using system calls or the "standard I/O package" (see *stdio*(3S)) for output during that time can cause unpredictable results.

## SEE ALSO

cc(1), ld(1), ioctl(2), plot(3X), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S), profile(4), term(4), terminfo(4), varargs(5).
termio(7), tty(7) in the *System Administrator's Reference Manual.*
Chapter 10 of the *Programmer's Guide*.

## NAME

directory: opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

## SYNOPSIS

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

DIR *opendir (filename)
char *filename;

struct dirent *readdir (dirp)
DIR *dirp;

long telldir (dirp)
DIR *dirp;

void seekdir (dirp, loc)
DIR *dirp;
long loc;

void rewinddir (dirp)
DIR *dirp;

void closedir(dirp)
DIR *dirp;
```

## DESCRIPTION

*Opendir* opens the directory named by *filename* and associates a *directory stream* with it. *Opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory, or if it cannot *malloc*(3X) enough memory to hold a DIR structure or a buffer for the directory entries.

*Readdir* returns a pointer to the next active directory entry. No inactive entries are returned. It returns NULL upon reaching the end of the directory or upon detecting an invalid location in the directory.

*Telldir* returns the current location associated with the named *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 from which *loc* was obtained was performed. Values returned by *telldir* are good only if the directory has not changed due to compaction or expansion. This is not a problem with System V, but it may be with some file system types.

*Rewinddir* resets the position of the named *directory stream* to the beginning of the directory.

*Closedir* closes the named *directory stream* and frees the DIR structure.

The following errors can occur as a result of these operations.

*opendir:*

| | |
|---|---|
| [ENOTDIR] | A component of *filename* is not a directory. |
| [EACCES] | A component of *filename* denies search permission. |
| [EMFILE] | The maximum number of file descriptors are currently open. |
| [EFAULT] | *Filename* points outside the allocated address space. |

*readdir:*

| | |
|---|---|
| [ENOENT] | The current file pointer for the directory is not located at a valid entry. |
| [EBADF] | The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed. |

*telldir, seekdir,* and *closedir:*

| | |
|---|---|
| [EBADF] | The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed. |

## EXAMPLE

Sample code which searches a directory for entry *name*:

```
dirp = opendir( "." );
while ( (dp = readdir( dirp )) != NULL )
        if ( strcmp( dp->d_name, name ) == 0 )
                {
                closedir( dirp );
                return FOUND;
```

(3X)

```
                    }
            closedir( dirp );
            return NOT_FOUND;
```

SEE ALSO

getdents(2), dirent(4).

WARNINGS

*Rewinddir* is implemented as a macro, so its function address cannot be taken.

(3X)

## NAME

getnum – calculate an integer value from a string of characters.

## SYNOPSIS

**int getnum** (string)
**char** *string;

## DESCRIPTION

*getnum* returns the integer value of a character string. *getnum* uses the following rules when calculating a number from a character string:

- Skip over any white space.
- Change a series of the numerical characters ( 0 - 9 ) into a number assuming base 10 representation.
- A series of numerical characters may end with **k**, **b**, or **w** to specify multiplication by 1024, 512, or 2 respectively;
- A pair of numbers may be separated by **x** or * to indicate a product of those two numbers.
- Use the **null** and **colon** as the termination characters.
- If an illegal character is encountered before a *termination* character, an error conditions exists and -1 is returned.

The program must be loaded with the disk access library **/usr/lib/access.a**.

NAME

getperms – read the *permissions* file

SYNOPSIS

**int getperms** (disk)
**struct usrdev** ∗disk;

DESCRIPTION

When *getperms* is invoked, the member *look_for* is used to find a match in the permissions file /etc/perms. When a match of either the *real_device* entry or the *alias* entry is found, *getperms* returns the structure filled with the contents of the matching line.

The program must be loaded with the disk access library **/usr/lib/libaccess.a**

```
struct usrdev
{
                char real_dev[ ] ;        /* block device to access */
                char look_for[ ] ;        /* an alternative name for the device */
                char mntpt[ ] ;           /* the default mount point */
                char fsize[ ] ;           /* maximum file system size on the device */
                char mkfs_dev[ ] ;        /* device to use for making file systems */
                char modes[ ] ;           /* access permissions */
                char pgm[ ] ;             /* format utitily to envoke */
                char fmt_dev[ ] ;         /* redundant; will go away in future versions */
};
```

FILES

**/usr/include/access.h**

SEE ALSO

*perms*(4).

## NAME

ldahread – read the archive header of a member of an archive file

## SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

## DESCRIPTION

If **TYPE(***ldptr***)** is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

*ldahread* returns **SUCCESS** or **FAILURE**. *ldahread* will fail if **TYPE(***ldptr***)** does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4), ar(4).

## NAME

ldclose, ldaclose – close a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>


int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

## DESCRIPTION

*Ldopen*(3X) and *ldclose* are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE(*ldptr*)** does not represent an archive file, *ldclose* will close the file and free the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE(*ldptr*)** is the magic number of an archive file, and if there are any more files in the archive, *ldclose* will reinitialize **OFFSET(*ldptr*)** to the file address of the next archive member and return **FAILURE**. The **LDFILE** structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

*Ldaclose* closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE(*ldptr*)**. *Ldaclose* always returns **SUCCESS**. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

(3X)

## NAME

ldfhread – read the file header of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

## DESCRIPTION

*ldfhread* reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

*ldfhread* returns **SUCCESS** or **FAILURE**. *ldfhread* will fail if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro **HEADER(***ldptr***)** defined in **ldfcn.h** [see ldfcn (4)]. The information in any field, *fieldname*, of the file header may be accessed using **HEADER(ldptr).fieldname**.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

## NAME

ldgetname – retrieve symbol name for common object file symbol table entry

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

## DESCRIPTION

*ldgetname* returns a pointer to the name associated with **symbol** as a string. The string is contained in a static buffer local to *ldgetname* that is overwritten by each call to *ldgetname*, and therefore must be copied by the caller if the name is to be saved.

*ldgetname* can be used to retrieve names from object files without any backward compatibility problems. *ldgetname* will return NULL (defined in **stdio.h**) for an object file if the name cannot be retrieved. This situation can occur:

- if the "string table" cannot be found,
- if not enough memory can be allocated for the string table,
- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to *ldgetname* that looks like a reference to a name in a nonexistent string table), or
- if the name's offset into the string table is past the end of the string table.

Typically, *ldgetname* will be called immediately after a successful call to *ldtbread* to retrieve the name associated with the symbol table entry filled by *ldtbread*.

The program must be loaded with the object file access routine library **libld.a.**

## SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

## NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>


int ldlread(ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlinit(ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

## DESCRIPTION

*ldlread* searches the line number entries of the common object file currently associated with *ldptr*. *ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

*Ldlinit* and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into the memory beginning at *linent*.

*ldlread*, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE**. *ldlread* will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *Ldlinit* will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *Ldlitem* will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

## NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

## DESCRIPTION

*ldlseek* seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

*Ldnlseek* seeks to the line number entries of the section specified by *sectname*.

*ldlseek* and *ldnlseek* return **SUCCESS** or **FAILURE**. *ldlseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnlseek* will fail if there is no section name corresponding with **sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

(3X)

## NAME

ldohseek – seek to the optional file header of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

## DESCRIPTION

*ldohseek* seeks to the optional file header of the common object file currently associated with *ldptr*.

*ldohseek* returns **SUCCESS** or **FAILURE**. *ldohseek* will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a.**

## SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

## NAME

ldopen, ldaopen – open a common object file for reading

## SYNOPSIS

**#include <stdio.h>**
**#include <filehdr.h>**
**#include <ldfcn.h>**

**LDFILE *ldopen (filename, ldptr)**
**char *filename;**
**LDFILE *ldptr;**

**LDFILE *ldaopen (filename, oldptr)**
**char *filename;**
**LDFILE *oldptr;**

## DESCRIPTION

*ldopen* and *ldclose*(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL**, then *ldopen* will open *filename* and allocate and initialize the **LDFILE** structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if **TYPE(*ldptr*)** is the archive magic number, *ldopen* will reinitialize the **LDFILE** structure for the next archive member of *filename.*

*ldopen* and *ldclose*(3X) are designed to work in concert. *Ldclose* will return **FAILURE** only when **TYPE(*ldptr*)** is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*(3X).

(3X)

```
/* for each filename to be processed */
ldptr = NULL;
do
{
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, *ldaopen* will open *filename* anew and allocate and initialize a new **LDFILE** structure, copying the **TYPE**, **OFFSET**, and **HEADER** fields from *oldptr*. *Ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return **NULL** if *filename* cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

## NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

## DESCRIPTION

*ldrseek* seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

*Ldnrseek* seeks to the relocation entries of the section specified by *sectname*.

*ldrseek* and *ldnrseek* return **SUCCESS** or **FAILURE**. *ldrseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnrseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

## NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

## SYNOPSIS

**#include <stdio.h>**
**#include <filehdr.h>**
**#include <scnhdr.h>**
**#include <ldfcn.h>**

**int ldshread (ldptr, sectindx, secthead)**
**LDFILE *ldptr;**
**unsigned short sectindx;**
**SCNHDR *secthead;**

**int ldnshread (ldptr, sectname, secthead)**
**LDFILE *ldptr;**
**char *sectname;**
**SCNHDR *secthead;**

## DESCRIPTION

*ldshread* reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

*Ldnshread* reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

*ldshread* and *ldnshread* return **SUCCESS** or **FAILURE**. *ldshread* will fail if *sectindx* is greater than the number of sections in the object file; *ldnshread* will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a.**

## SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

## NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

## SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

## DESCRIPTION

*ldsseek* seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

*Ldnsseek* seeks to the section specified by *sectname*.

*ldsseek* and *ldnsseek* return **SUCCESS** or **FAILURE**. *ldsseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnsseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

## SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

## NAME

ldtbindex – compute the index of a symbol table entry of a common object file

## SYNOPSIS

**#include <stdio.h>**
**#include <filehdr.h>**
**#include <syms.h>**
**#include <ldfcn.h>**

**long ldtbindex (ldptr)**
**LDFILE *ldptr;**

## DESCRIPTION

*ldtbindex* returns the **(long)** index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it will return the index of the next entry.

*ldtbindex* will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a.**

## SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

*ldtbread* reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at **symbol**.

*ldtbread* returns **SUCCESS** or **FAILURE**. *ldtbread* will fail if *symindex* is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a.**

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldgetname(3X), ldfcn(4).

## NAME

ldtbseek – seek to the symbol table of a common object file

## SYNOPSIS

**#include <stdio.h>**
**#include <filehdr.h>**
**#include <ldfcn.h>**

**int ldtbseek (ldptr)**
**LDFILE *ldptr;**

## DESCRIPTION

*ldtbseek* seeks to the symbol table of the common object file currently associated with *ldptr*.

*ldtbseek* returns **SUCCESS** or **FAILURE**. *ldtbseek* will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a.**

## SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

## NAME

logname – return login name of user

## SYNOPSIS

**char *logname( )**

## DESCRIPTION

*logname* returns a pointer to the null-terminated login name; it extracts the **LOGNAME** environment variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

## FILES

/etc/profile

## SEE ALSO

getenv(3C), profile(4), environ(5).
env(1), login(1) in the *User's Reference Manual.*

## CAVEATS

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

This method of determining a login name is subject to forgery.

(3X)

## NAME

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

## SYNOPSIS

```
#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

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

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

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo()
```

## DESCRIPTION

*malloc* and *free* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "–lmalloc" is used with *cc*(1) or *ld*(1).

*malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, and its contents have been destroyed (but see *mallopt* below for a way to change this behavior).

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *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.

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

*Mallopt* provides for control over the allocation algorithm. The available values for *cmd* are:

| | |
|---|---|
| M_MXFAST | Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24. |
| M_NLBLKS | Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 0. The default value for *numlblks* is 100. |
| M_GRAIN | Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set. |
| M_KEEP | Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended. |

These values are defined in the <*malloc.h*> header file.

*Mallopt* may be called repeatedly, but may not be called after the first small block is allocated.

*Mallinfo* provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo {
        int arena;          /* total space in arena */
        int ordblks;        /* number of ordinary blocks */
        int smblks;         /* number of small blocks */
        int hblkhd;         /* space in holding block headers */
        int hblks;          /* number of holding blocks */
        int usmblks;        /* space in small blocks in use */
        int fsmblks;        /* space in free small blocks */
        int uordblks;       /* space in ordinary blocks in use */
        int fordblks;       /* space in free ordinary blocks */
        int keepcost;       /* space penalty if keep option */
                            /* is used */
}
```

This structure is defined in the <*malloc.h*> header file.

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

brk(2), malloc(3C).

## DIAGNOSTICS

*malloc, realloc* and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

## WARNINGS

This package usually uses more data space than *malloc*(3C).
The code size is also bigger than *malloc*(3C).
Note that unlike *malloc*(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.
Undocumented features of *malloc*(3C) have not been duplicated.

(3X)

## NAME

regcmp, regex – compile and execute regular expression

## SYNOPSIS

```
char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;
```

## DESCRIPTION

*regcmp* compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. *Malloc*(3C) is used to create space for the compiled form. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument. *regcmp*(1) has been written to generally preclude the need for this routine at execution time.

*Regex* executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^ These symbols retain their meaning in *ed*(1).

$ Matches the end of the string; **\n** matches a new-line.

– Within brackets the minus means *through*. For example, **[a–z]** is equivalent to **[abcd...xyz]**. The – can appear as itself only if used as the first or last character. For example, the character class expression **[]–]** matches the characters **]** and **–**.

+ A regular expression followed by + means *one or more times*. For example, **[0–9]+** is equivalent to **[0–9] [0–9]***.

{m} {m,} {m,u}
Integer values enclosed in {} indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The

value {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

( . . . )$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

( . . . ) Parentheses are used for grouping. An operator, e.g., *, +, {}, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped with a \ (backslash) to be used as themselves.

**EXAMPLES**

Example 1:

```
        char *cursor, *newcursor, *ptr;
                . . .
        newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
        free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
        char ret0[9];
        char *newcursor, *name;
                . . .
        name = regcmp("([A–Za–z][A–za–z0–9]{0,7})$0", (char *)0);
        newcursor = regex(name, "012Testing345", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (the "4"). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
        #include "file.i"
        char *string, *newcursor;
                . . .
        newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** [see *regcmp*(1)] against *string*.

These routines are kept in **/lib/libPW.a**.

SEE ALSO

regcmp(1), malloc(3C).
ed(1) in the *User's Reference Manual.*

BUGS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

(3X)

## NAME

sputl, sgetl – access long integer data in a machine-independent fashion.

## SYNOPSIS

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

## DESCRIPTION

*sputl* takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

*Sgetl* retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be loaded with the object-file access routine library **libld.a.**

(3X)