Chapter 3

# Subroutines and Libraries

This chapter describes functions conventionally found in various libraries, other than those functions that directly invoke X/OPEN system calls, which are described in Chapter 2 of this part of the Guide. Certain major collections are identified by a letter after the section number. The established 3C, 3M, 3S, and 3X nomenclature is preserved, but is not important in the context of an interface definition.

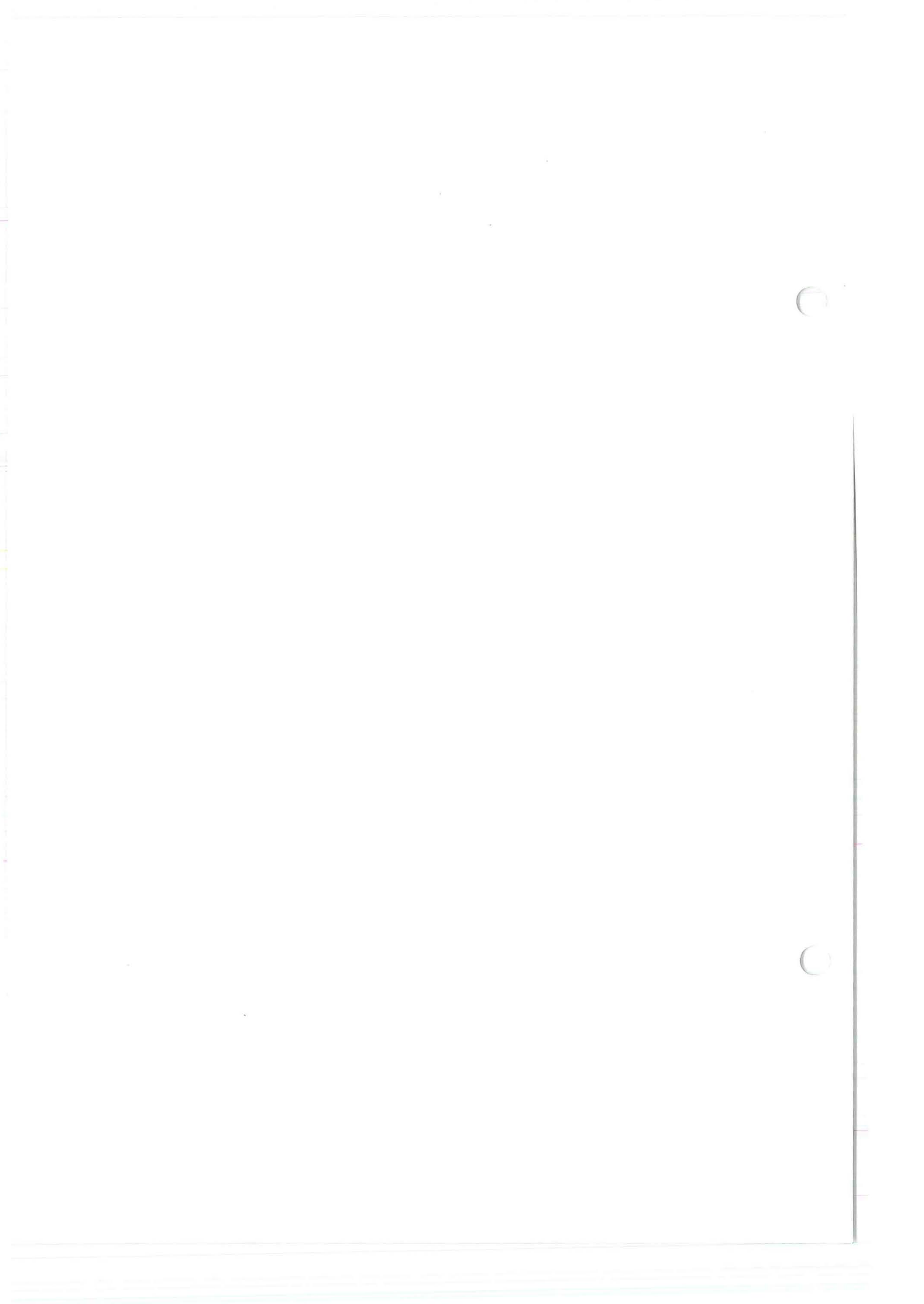
(3C) These functions, together with those of chapter 2 and those marked (3S), constitute the conventional C Library. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(3M) These functions constitute the *optional* math group. Declarations for these functions may be obtained from the **#include** file **<math.h>**.

(3S) These functions constitute the "standard I/O group" (see *stdio*(3S)). Declarations for these functions may be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialised functions.

Functions in the math group (3M) may return the conventional values **0** or HUGE (the largest single-precision floating-point number) when the function is undefined for the given arguments or when the value is not representable. In these cases, the external variable *errno* is set to the value [EDOM] or [ERANGE].



## NAME

abort — generate an abnormal process abort

## SYNOPSIS

int abort ()

## DESCRIPTION

*Abort* first closes all open files if possible then causes the process abort signal, SIGABRT, to be sent to the process. This invokes abnormal process termination routines, such as a *core dump*, which are implementation dependent.

## ERRORS

None.

## APPLICATION USAGE

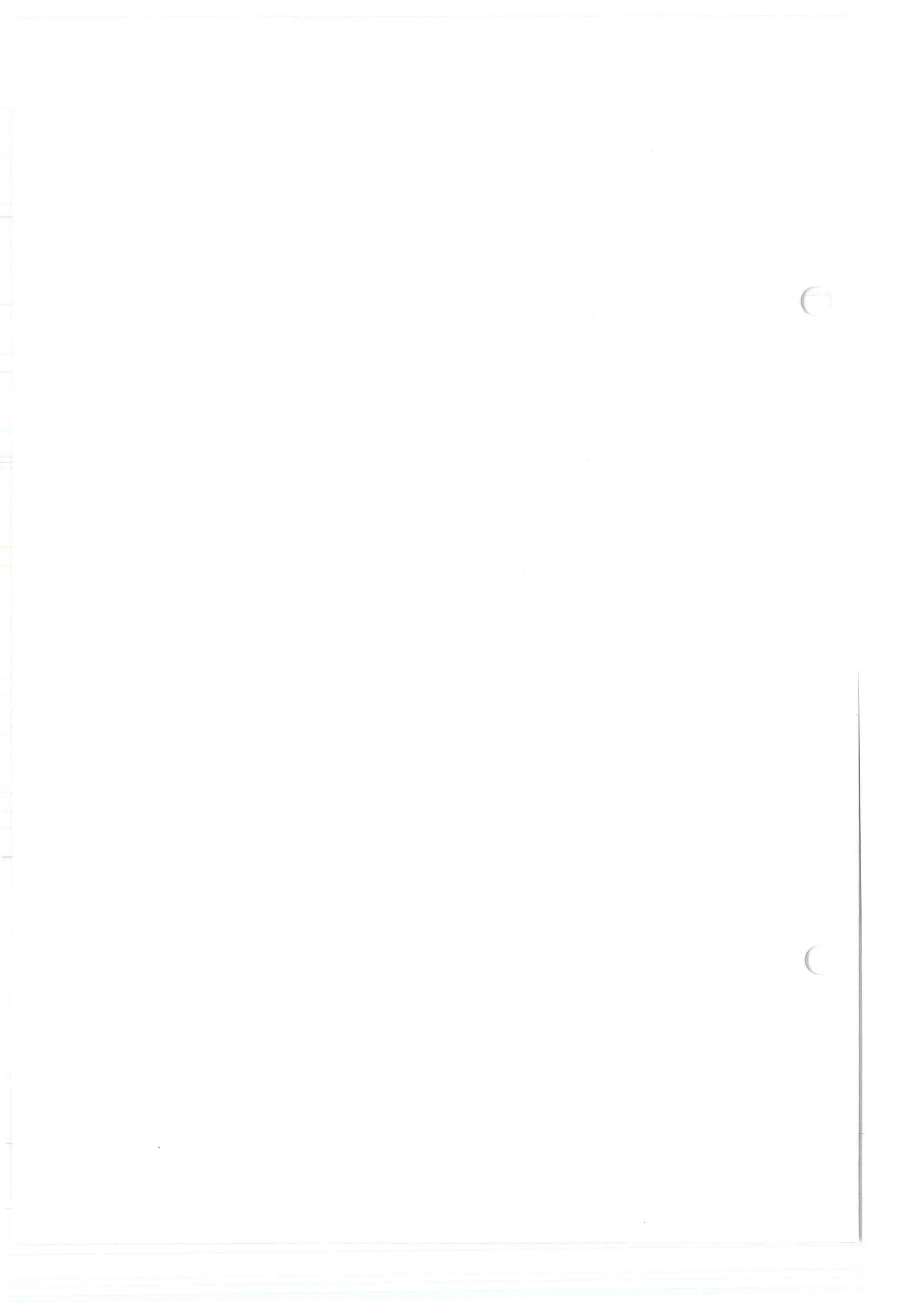
SIGABRT is not intended to be caught.

## SEE ALSO

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

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the signal is identified as SIGABRT. The SVID does not identify the signal, but forecasts in the FUTURE DIRECTIONS section that it will be SIGABRT. (In UNIX System V Release 2.0 it is SIGIOT.)



## NAME

abs — return integer absolute value

## SYNOPSIS

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

## DESCRIPTION

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

## SEE ALSO

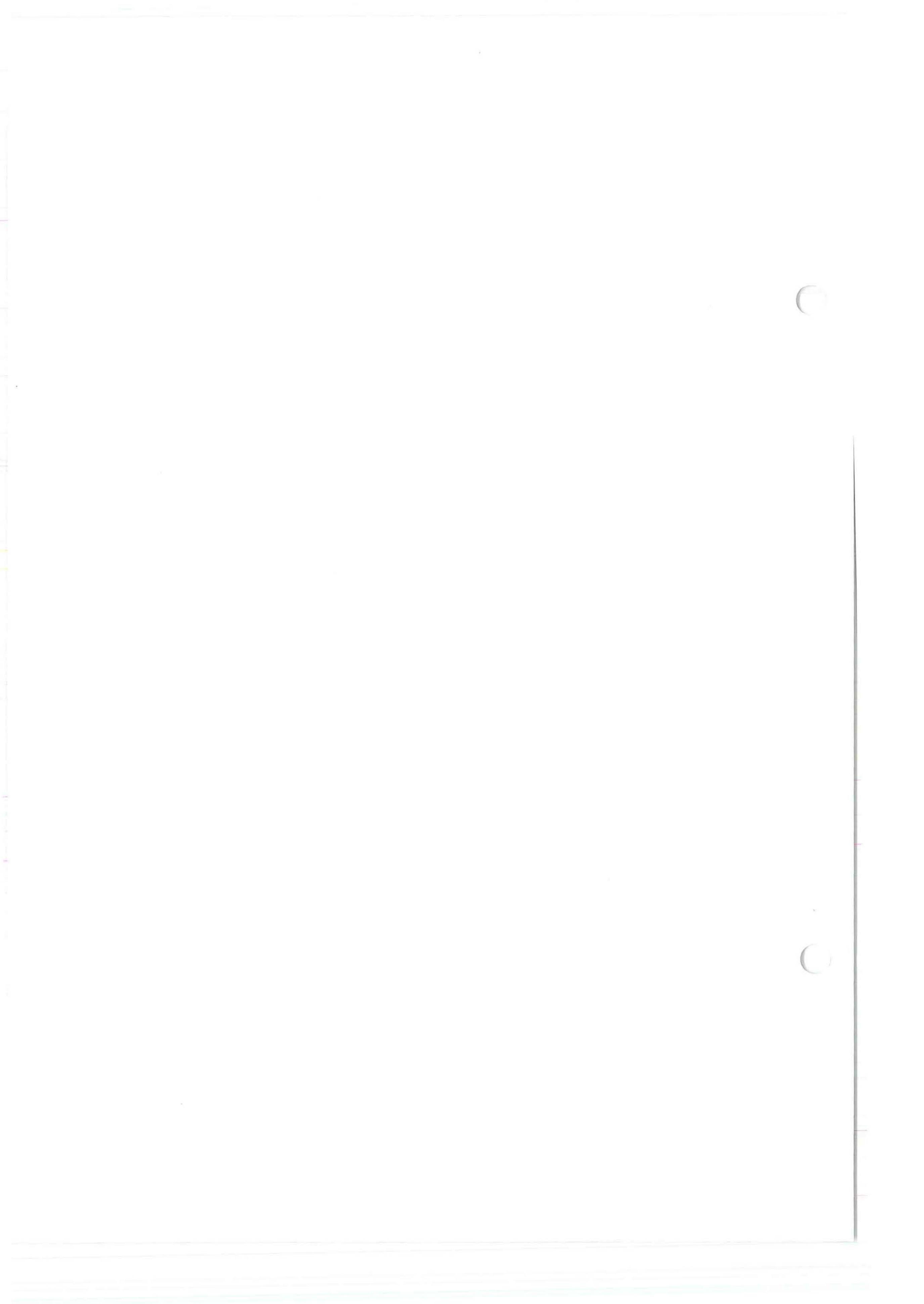
fabs(3C).

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.



## NAME

assert — verify program assertion

## SYNOPSIS

```
#include <assert.h>

void 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.

## SEE ALSO

abort(3C).

## APPLICATION USAGE

Forcing a definition of the name *NDEBUG,* either from the compiler command line or with the preprocessor control statement *#define NDEBUG* ahead of the *#include <assert.h>* statement, will stop assertions from being compiled into the program.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

j0, j1, jn, y0, y1, yn — Bessel functions OPTIONAL

## 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.

## RETURN VALUE

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).

## SEE ALSO

matherr(3M).

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in the SVID.

## NAME

bsearch — binary search a sorted table

## SYNOPSIS

```
#include <search.h>

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

## DESCRIPTION

*Bsearch* is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function, *compar*. *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. *Width* is the size of an element in bytes. *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.

## RETURN VALUE

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

## SEE ALSO

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

## APPLICATION USAGE

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

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>

#define TABSIZE        1000

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

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that a programming error in the *node_compare* part of the example has been corrected.

## 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) or *system*(3S).

## SEE ALSO

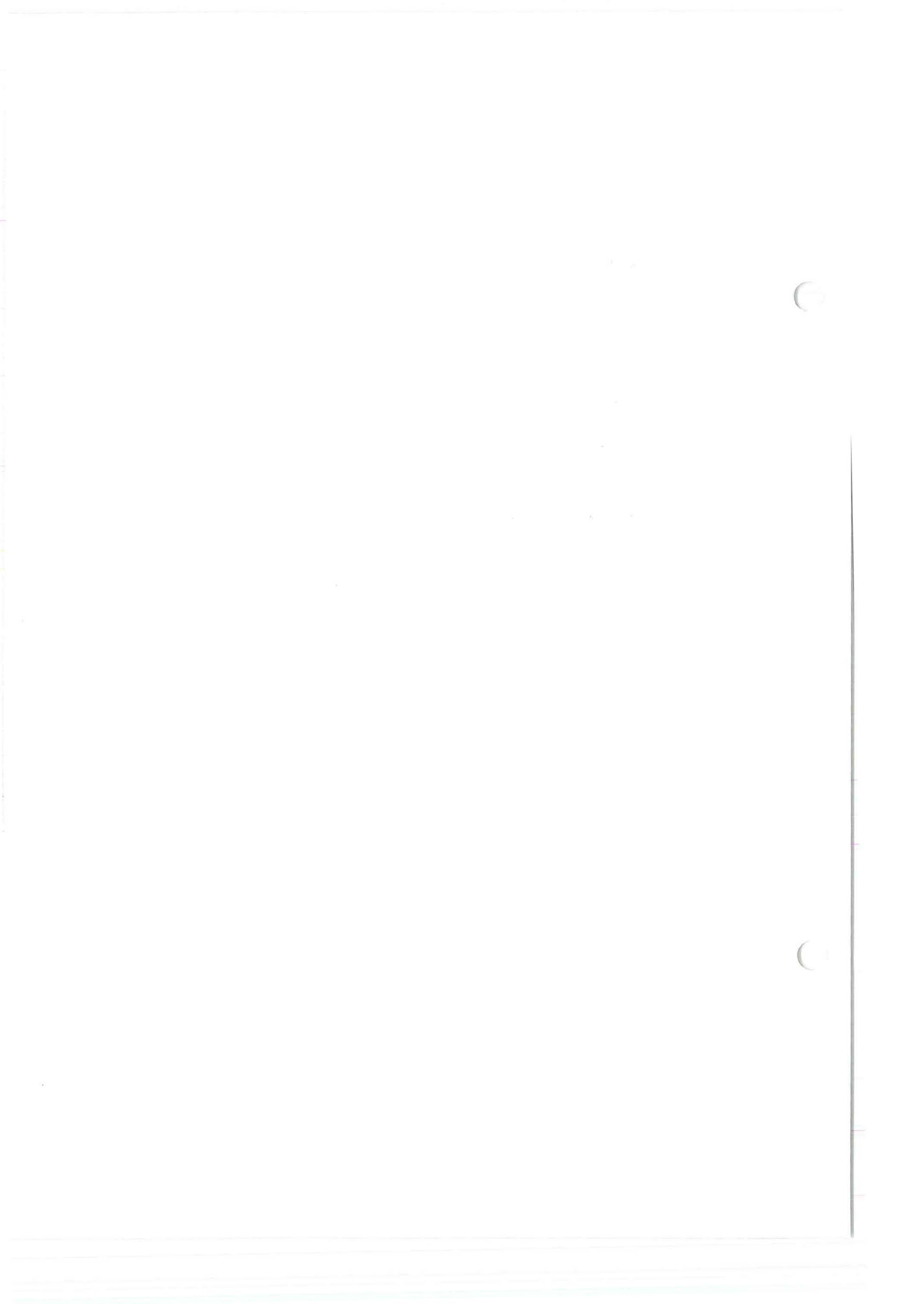
times(2), wait(2).

## APPLICATION USAGE

The value returned by *clock* is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution.

## RELATIONSHIP TO SVID

Identical to the SVID entry.



## NAME

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. The macro *_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).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

crypt, setkey, encrypt — generate DES encryption

## SYNOPSIS

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

void setkey (key)
char *key;

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

## DESCRIPTION

*Crypt* is the password encryption function. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

*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 DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the *salt* itself.

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

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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, with contents overwritten at the next call to *ctermid*, and returned address. 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.

## SEE ALSO

ttyname(3C).

## APPLICATION USAGE

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 (such as **/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.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the words "such as" have been inserted before "/dev/tty" in the APPLICATION USAGE section.

## NAME

ctime, localtime, gmtime, asctime, tzset, timezone, daylight, tzname — convert date and time to string

## SYNOPSIS

```
#include <time.h>

#include <sys/types.h>

char *ctime (clock)
long *clock;

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *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:

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

All the fields have constant width. *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 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 contains the following fields:

```
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 is 0) */
int tm_yday;    /* day of year (0 - 365) */
int tm_isdst;   /* Daylight savings time flag */
```

*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 MET, *timezone* is -1*60*60); the external variable *daylight* is non-zero if and only if some Daylight Savings Time conversion should be applied.

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 an optional minus sign (for zones east of Greenwich) and a series of digits representing the difference between local time and GMT in hours; this is followed by an optional three letter name for a daylight time zone. The setting for most of continental Europe would be MET-1EET. 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] = {"MET", "EET"};
```

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.

## SEE ALSO

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

## APPLICATION USAGE

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

The usefulness of these functions is diminished by the fact that they are inappropriate in those parts of the world which do not base their reckoning of time upon **GMT**. The algorithm used to determine whether Daylight Savings Time applies depends on the location in question; as it is usually supplied, the implementation of *ctime* only knows about a small number of the necessary conversions. There is no agreed international standard for timezone names. The following names are suggested for Europe:

| | | |
|---|---|---|
| WET | (GMT) | Western European Time, eg U.K. |
| MET | (GMT+1) | Middle European Time |
| EET | (GMT+2) | Eastern European Time |

## FUTURE DIRECTIONS

The argument *clock* to *ctime*, *localtime*and *gmtime* will be declared through the **typedef** facility as a pointer to *time_t*.

The number in TZ will be defined as an optional minus sign followed by two hour digits and two minute digits, *[-]hhmm*, to represent fractional timezones.

## RELATIONSHIP TO SVID

Identical to SVID, except that European timezone names have replaced references to North America, and the names *timezone*, *daylight* and *tzname* have been added to the NAME section.

In the second paragraph of the DESCRIPTION section, the SVID uses the term "UNIX system" instead of "system".

## NAME

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. Each is a predicate returning non-zero for true, zero (0) for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF, see *stdio*(3S).

*isalpha* c is a letter.

*isupper* c is an upper-case letter.

*islower* c is a lower-case letter.

*isdigit* c is a digit [0-9].

*isxdigit* c is a 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, new-line, vertical tab or form-feed.

*ispunct* c is a punctuation character (neither control nor alphanumeric).

*isprint* c is a printing character, code 040 (space) through 0176 (tilde).

*isgraph* c is a printing character, like *isprint* except false for space.

*iscntrl* c is a delete character (0177) or an ordinary control character (less than 040).

*isascii* c is an ASCII character, code between 0 and 0177 inclusive.

## RETURN VALUE

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

## APPLICATIONS USAGE

To ensure applications portability, especially across natural languages, no other character classification method should be used.

SEE ALSO

stdio(3S).

RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

curses — CRT screen handling and optimization package (OPTIONAL)

## SYNOPSIS

**#include <curses.h>**

## DESCRIPTION

Following is a list of the *minicurses* library functions found in System V, Release 2.0. This is for information only, and there is no guarantee that a particular X/OPEN system will offer support for this package. Chapter 1 describes the X/OPEN attitude towards *curses*. The header file **<curses.h>** is *not* a part of the XVS.

This package is usually implemented via a database of terminal capabilities. Some systems use a human-readable form called *termcap*, while others use a compiled form known as *terminfo*. Neither of these databases are described in the XVS.

## FUNCTIONS

| | |
|---|---|
| addch(ch) | add a character to *stdscr* (like putchar) (wraps to next line at end of line) |
| addstr(str) | calls addch with each character in *str* |
| attroff(attrs) | turn off attributes named |
| attron(attrs) | turn on attributes named |
| attrset(attrs) | set current attributes to *attrs* |
| baudrate( ) | current terminal speed |
| beep( ) | sound beep on terminal |
| cbreak( ) | set cbreak mode |
| delay_output(ms) | insert *ms* millisecond pause in output |
| echo( ) | set echo mode |
| endwin( ) | end window modes |
| flushinp( ) | throw away any typeahead |
| getch( ) | get a char from tty |
| idlok(win, bf) | use terminal's insert/delete line if *bf* != 0 |
| initscr( ) | initialize screens |
| meta(win, flag) | allow meta characters on input if *flag* != 0 |
| move(y, x) | move to (*y*, *x*) on *stdscr* |
| nl( ) | set newline mapping |
| nocbreak( ) | unset cbreak mode |
| noecho( ) | unset echo mode |
| nonl( ) | unset newline mapping |
| noraw( ) | unset raw mode |
| raw( ) | set raw mode |
| refresh( ) | make current screen look like *stdscr* |
| resetterm( ) | set tty modes to "out of curses" state |
| resetty( ) | reset tty flags to stored value |
| saveterm( ) | save current modes as "in curses" state |

| | |
|---|---|
| savetty() | store current tty flags |
| standend() | clear standout mode attribute |
| standout() | set standout mode attribute |
| unctrl(ch) | printable version of *ch* |

## RELATIONSHIP TO SVID

The SVID, in Section 5 FUTURE DIRECTIONS (5.1 User Interface Services Extension), says that applications relying on *curses* will be compatible with whatever is done to the User Interface Services Extension at a future date. In Section 4 EXT: OTHER PLANNED EXTENSIONS, oblique mention is made of *curses* by referring to the System V Release 2.0 *libcurses* library. No detail of its contents is given.

## NAME

cuserid — character login name of the user

## SYNOPSIS

```
#include <stdio.h>

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

## DESCRIPTION

*Cuserid* generates a character representation of the login name of the owner of the current process. If *s* is NULL, this representation is generated in an internal static area, the address of which is returned. If *s* is not NULL, *s* is assumed to point to an array of at least {L_cuserid} characters; the representation is left in this array. The symbolic constant {L_cuserid} is defined in **<stdio.h>**.

## RETURN VALUE

If *s* is NULL and the login name cannot be found, *cuserid* returns NULL; if *s* is not NULL and the login name cannot be found, the null character '\0' will be placed at **s*.

## SEE ALSO

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

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## 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.

N.B. In the following, the formal mathematical notation [0.0, 1.0) indicates an interval including 0.0 but not including 1.0.

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

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n>0.25n'=0.$$

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; that is why they 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. That is why these routines do not have to be initialized; the calling program merely has to 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 {LONG_BIT} 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.

## SEE ALSO

rand(3C).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the SVID paragraph in the APPLICATIONS USAGE section has been moved up into the DESCRIPTION section and added onto the end of the *seed48* paragraph.

## 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 (0). 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.

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

end, etext, edata — last locations in program (OPTIONAL)

## 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*(3X), standard input/output (*stdio*(3S)), and so on. Thus, the current value of the program break should be determined by *sbrk(0)*, see *brk*(2).

## SEE ALSO

brk(2), malloc(3X), stdio(3S).

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

erf, erfc — error function and complementary error functions (OPTIONAL)

## SYNOPSIS

```
#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;
```

## DESCRIPTION

*Erf* returns the error function of *x*, defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

*Erfc* returns 1.0 — *erf*(*x*).

## SEE ALSO

exp(3M).

## APPLICATION USAGE

*Erfc* 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.

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in SVID.

## NAME

exp, log, log10, pow, sqrt — exponential, logarithm, power, square root functions (OPTIONAL)

## 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 (0), *y* must be positive. If *x* is negative, *y* must be an integer value.

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

## RETURN VALUE

*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).

## SEE ALSO

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

## FUTURE DIRECTIONS

A macro HUGE_VAL will be defined in the header file <math.h>. This macro will call a function which will either return $+\infty$ on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.
If the correct value overflows, *exp* will return HUGE_VAL.

*Log* and *log10* will return -HUGE_VAL when *n* is not positive.

*Sqrt* will return -0 when the value of *n* is -0.

The return value of *pow* will be negative HUGE_VAL when an illegal combination of input arguments is passed to *pow*.

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in SVID.

## 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.

## RETURN VALUE

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.

## SEE ALSO

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 (0).

*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).

## SEE ALSO

open(2), fopen(3S).

## APPLICATIONS USAGE

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

floor, ceil, fmod, fabs — floor, ceiling, remainder, absolute value functions (OPTIONAL)

## 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(3M).

## FUTURE DIRECTIONS

*Fmod* will return $x$ if $y$ is zero or if $x/y$ would overflow.

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in SVID.

## NAME

fopen, freopen, fdopen — open a stream

## SYNOPSIS

```
#include <stdio.h>

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

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

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

## DESCRIPTION

*Fopen* opens the file named by *file-name* and associates a *stream* with it. *Fopen* returns a pointer to the *FILE* structure associated with the *stream*.

*File-name* 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*(2), *dup*(2), *creat (2)* 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.

## RETURN VALUE

*Fopen* and *freopen* return a NULL pointer if *file-name* cannot be accessed and the external variable *errno* may contain any of the values listed for *open*(2).

## SEE ALSO

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

fread, fwrite — binary input/output

## SYNOPSIS

```
#include <stdio.h>

#include <sys/types.h>

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

int fwrite (ptr, size, nitems, stream)
char *ptr;
size_t size;
int nitems;
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*. *Fwrite* increments the file pointer in *stream* , if defined, by the number of bytes written.

## RETURN VALUE

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

## SEE ALSO

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

## APPLICATION USAGE

The argument *size* is typically *sizeof(*ptr)* where the C operator **sizeof** gives 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**.

*Ferror*(3S) or *feof*(3S) must be used to distinguish between an error condition and an end-of-file condition.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the parameters of *fread* and *fwrite* that are declared to be of type *size_t* are of type **int** in the SVID. Also, to define this type the header file <**sys/types.h**> is specified for inclusion. This is a FUTURE DIRECTION in the SVID.

## 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 $0.5 = |x| < 1.0$, 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 (0), 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*.

## RETURN VALUE

If *ldexp* would cause overflow, ±HUGE is returned (according to the sign of *value*), and *errno* is set to [ERANGE].
If *ldexp* would cause underflow, 0 is returned and *errno* is set to [ERANGE].

## FUTURE DIRECTIONS

A macro HUGE_VAL will be defined in the header file **<math.h>** . This macro will call a function which will either return $-\infty$ on a system supporting the IEEE P754 standard, or +{MAXDOUBLE} on a system which does not.

The return value *ldexp* will be ±HUGE_VAL (according to the sign of *value*) in case of overflow.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

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

## SYNOPSIS

```
#include <unistd.h>
#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, respectively, as *ptrname* takes the value SEEK_SET, SEEK_CUR or SEEK_END †.

*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*. The offset is always measured in bytes.

## SEE ALSO

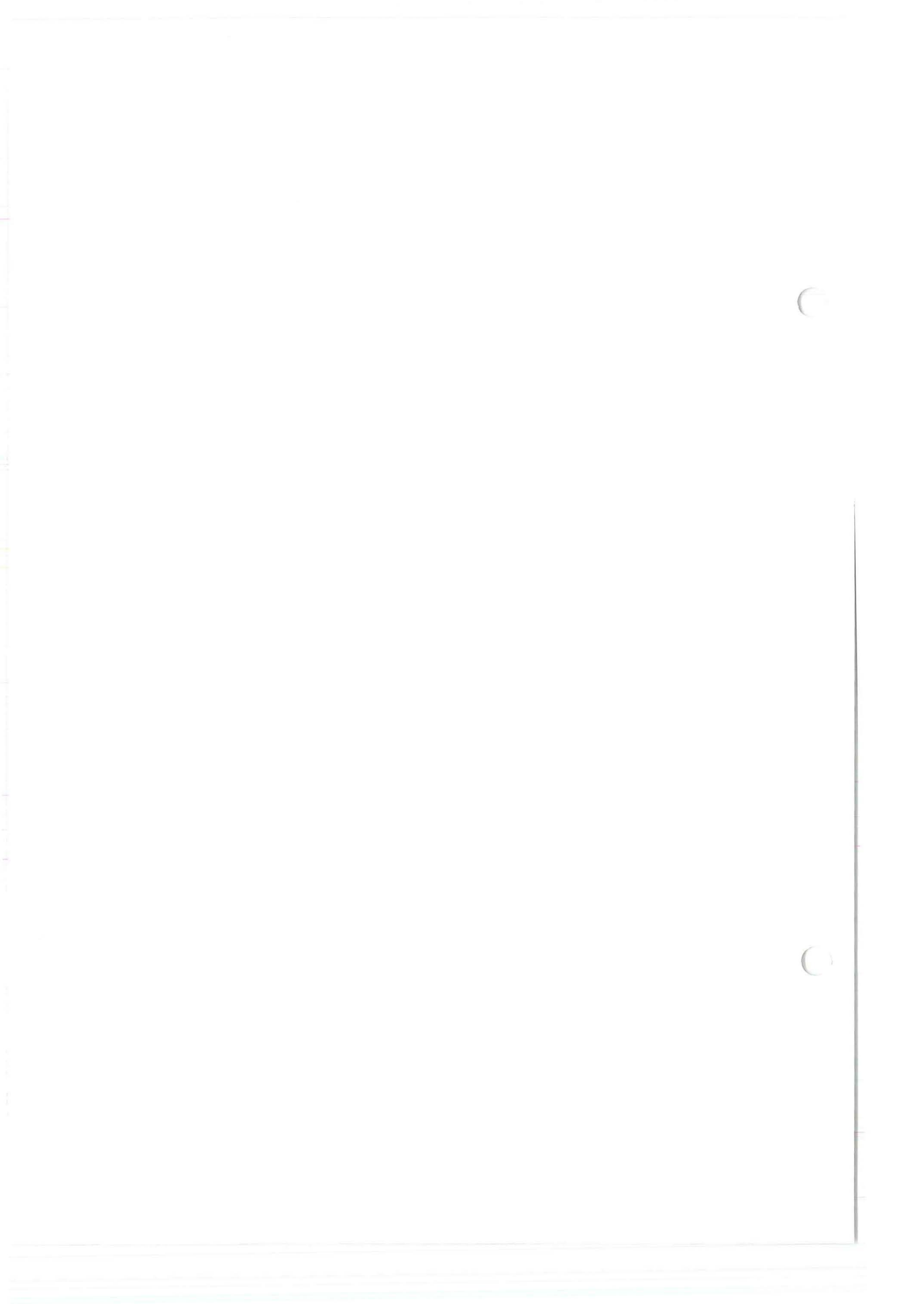
lseek(2), fopen(3S), ungetc(3S).

## RETURN VALUE

*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*; particularly, *fseek* may not be used on a terminal, or a file opened via *popen*(3S)‡.

## RELATIONSHIP TO SVID

† SVID does not use symbolic values - SEEK_SET, SEEK_CUR and SEEK_END for *ptrname*.
Wording change at end of Return Value. The SVID reads "...via *fopen*; particularly, *fseek* may not be used on a terminal or a file opened vi *popen*(OS).



## 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 non-zero value, or some error is detected within *ftw* (such as an I/O error). If the tree is exhausted, *ftw* returns zero (0). If *fn* returns a non-zero 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(3X).

## APPLICATION USAGE

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

*Ftw* uses *malloc*(3X) 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 non-zero value at its next invocation.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

gamma, signgam — log gamma function (OPTIONAL)

## SYNOPSIS

```
#include <math.h>

double gamma (x)
double x;

extern int signgam;
```

## DESCRIPTION

*Gamma* returns $\ln(|\Gamma(x)|)$, where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t} t^{x-1} dt$. 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 $\Gamma$:

```
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.

## RETURN VALUE

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).

## SEE ALSO

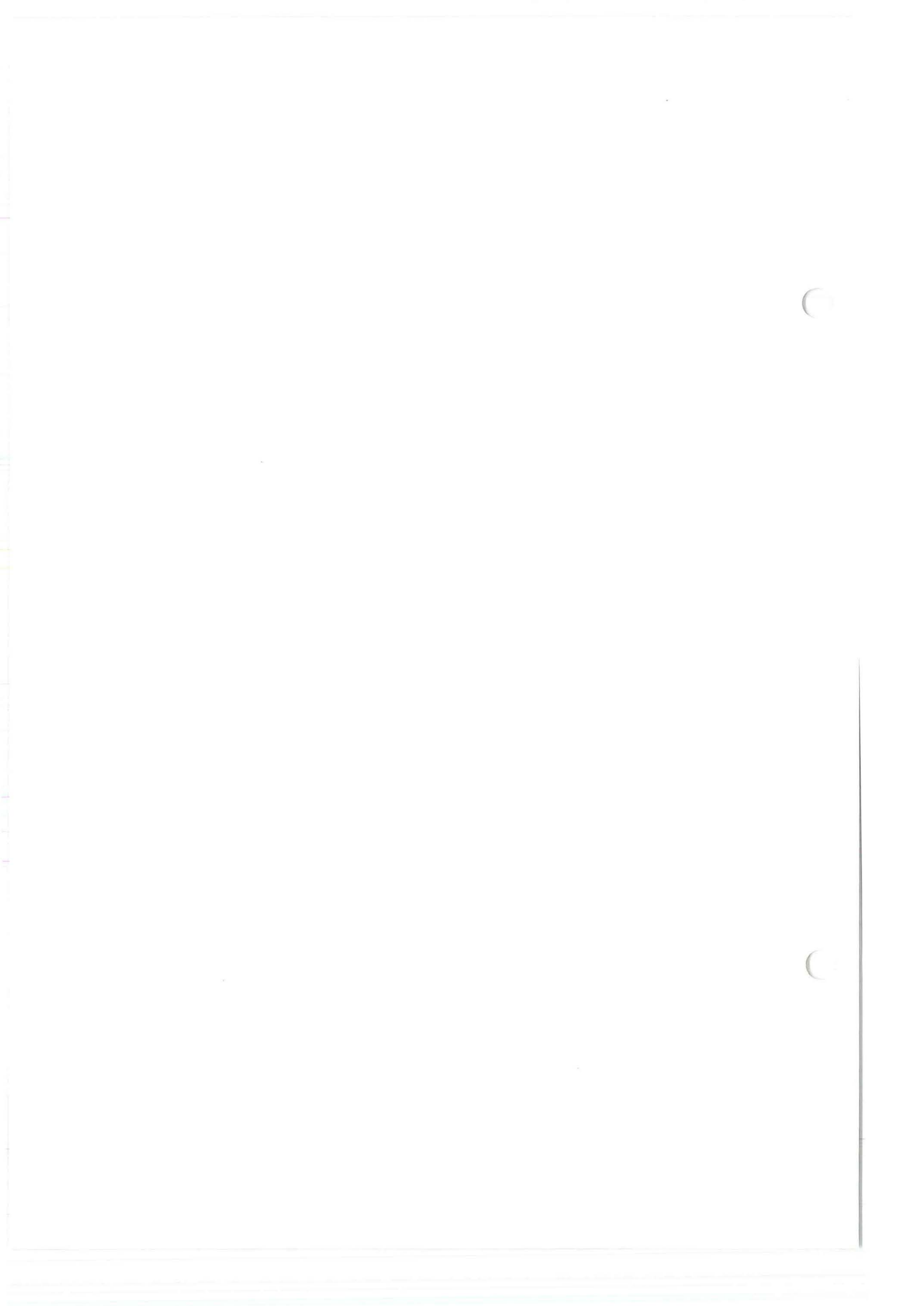
exp(3M), matherr(3M).

## FUTURE DIRECTIONS

A macro HUGE_VAL will be defined in the header file **<math.h>**. This macro will call a function which will either return $+\infty$ on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.
If the correct value overflows, *gamma* will return HUGE_VAL.

## RELATIONSHIP TO SVID

Identical to the SVID entry, with the addition of *signgam* to the NAME section. The *optional* mathematical group is mandatory in SVID.



## 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.

## RETURN VALUE

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.

## SEE ALSO

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

## APPLICATION USAGE

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.

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.

Because it is implemented as a macro, *getc* treats incorrectly a *stream* argument with side effects. In particular, *getc (*f++)* does not work sensibly. *Fgetc* should be used instead.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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*(3X). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free.*

## ERRORS

| | |
|---|---|
| [EINVAL] | *size* is zero |
| [ENOMEM] | no space available |
| [ERANGE] | *size* not large enough to hold the path-name. |

## RETURN VALUE

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

## SEE ALSO

malloc(3X).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

getenv — return value for environment name

## SYNOPSIS

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

## DESCRIPTION

*Getenv* searches the environment list 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 is returned.

## SEE ALSO

exec(2), putenv(3C).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

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

## SYNOPSIS

```
#include <grp.h>

struct group *getgrent ();

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

int setgrent ();

int endgrent ();
```

## DESCRIPTION

*Getgrent*, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

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

The members of this structure are:

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

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

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

## FILES

/etc/group

## SEE ALSO

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

## DIAGNOSTICS

A NULL pointer is returned on EOF or error.

### APPLICATION USAGE

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

### RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## 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*(3C) to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to call *cuserid*(3S), or to call *getlogin* and if it fails, to call *getpwuid*(3C).

## FILES

/etc/utmp

## SEE ALSO

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

## DIAGNOSTICS

Returns NULL if name not found.

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

getopt — get option letter from argument vector

## SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char *argv [], *optstring;

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

## DESCRIPTION

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

*Getopt* places in *optind* the *argv* index of the next argument to be processed. The external variable *optind* 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 EOF. The special option -- may be used to delimit the end of the options; EOF will be returned, and -- will be skipped.

## RETURN VALUE

*Getopt* prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*. This error message may be disabled by setting *opterr* to zero (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 options **f** and **o**, both of which require arguments:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        int bflg, aflg, errflg;
        char *ifile;
        char *ofile;
        extern char *optarg;
        extern int optind;
```

```
	.
	.
	while ((c = getopt(argc, argv, "abf:o:")) != EOF)
		switch (c) {
		case 'a':
			if (bflg)
				errflg++;
			else
				aflg++;
			break;
		case 'b':
			if (aflg)
				errflg++;
			else
				bproc( );
			break;
		case 'f':
			ifile = optarg;
			break;
		case 'o':
			ofile = optarg;
			break;
		case '?':
			errflg++;
		}
	if (errflg) {
		fprintf(stderr, "usage: . . . ");
		exit (2);
	}
	for ( ; optind < argc; optind++) {
		if (access(argv[optind], 4)) {
	.
	.
	.
}
```

## FUTURE DIRECTIONS

*getopt* will be enhanced to enforce all rules of the System V Command Syntax Standard.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

getpass — read a password

## SYNOPSIS

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

## DESCRIPTION

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

## FILES

/dev/tty

## SEE ALSO

crypt(3C).

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

getpw — get name from UID

## SYNOPSIS

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

## DESCRIPTION

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

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

## FILES

/etc/passwd

## SEE ALSO

getpwent(3C), passwd(5).

## DIAGNOSTICS

Non-zero return on error.

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

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

## SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ();

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

int setpwent ();

int endpwent ();
```

## DESCRIPTION

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

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

struct comment
        char *c_dept;
        char *c_name;
        char *c_acct;
        char *c_bin:
```

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

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

encountered.

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

## FILES

/etc/passwd

## SEE ALSO

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

## DIAGNOSTICS

Null pointer returned on EOF or error.

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## 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.

## RETURN VALUE

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.

## SEE ALSO

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

## APPLICATION USAGE

Reading a line that is too long through *gets* causes *gets* to break. The use of *fgets* is recommended.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

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 containing the following members:

*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  ut_exit;        /* The exit status of a process
                                     * marked as DEAD_PROCESS. */
time_t              ut_time;        /* time entry made */
```

*struct exit_status* contains

```
short   e_termination;   /* Process termination status */
short   e_exit;          /* Process exit
```

*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 *ut_type* 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 doesn't 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(5).

## 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.

## APPLICATION USAGE

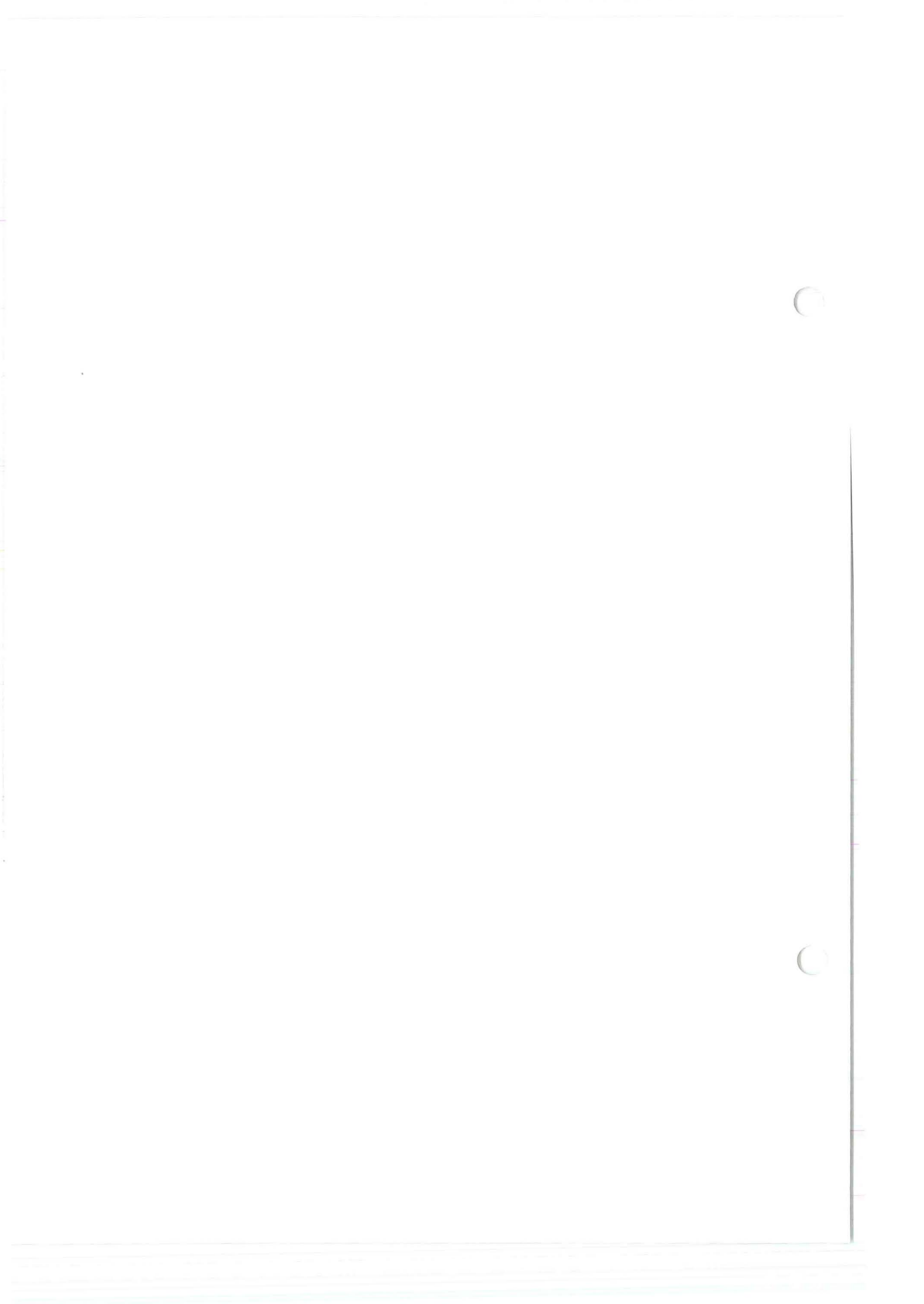
The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what 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 isn't 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.

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.



## 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*.

## RETURN VALUE

*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 (0) if it cannot allocate sufficient space for the table.

## SEE ALSO

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

## APPLICATION USAGE

*Hsearch* and *hcreate* use *malloc*(3X) to allocate space. *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 manner 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;
```

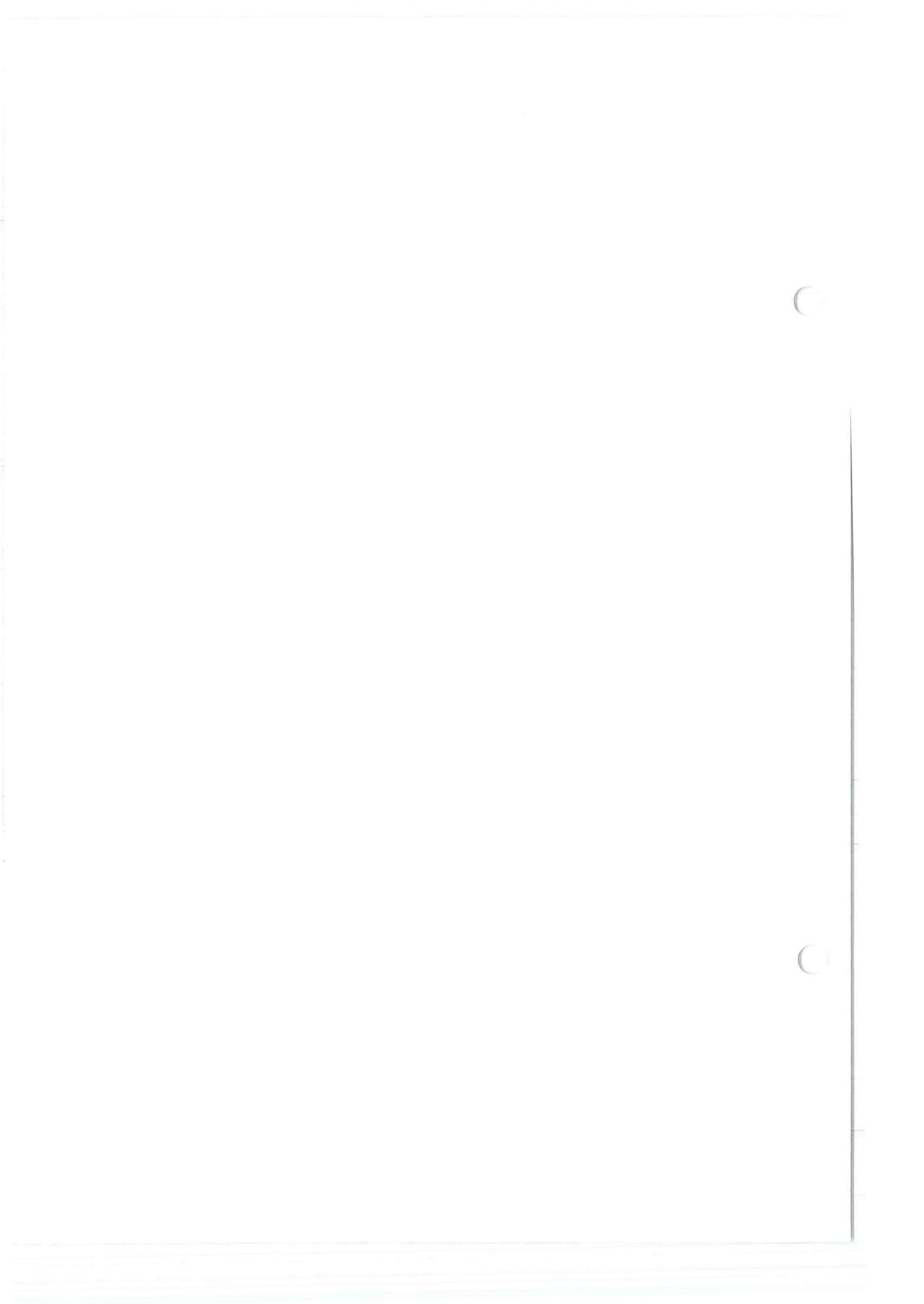
```
        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */
        char name_to_find[30];
        int i = 0;

        /* 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)
            }
        }
}
```

## FUTURE DIRECTIONS

The restriction of having only one hash search table active at any given time will be removed.

## RELATIONSHIP TO SVID

Identical to the SVID entry.



## NAME

hypot — Euclidean distance function (OPTIONAL)

## SYNOPSIS

```
#include <math.h>

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

## DESCRIPTION

*Hypot* returns

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

taking precautions against unwarranted overflows.

## RETURN VALUE

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).

## SEE ALSO

matherr(3M).

## FUTURE DIRECTIONS

A macro HUGE_VAL will be defined in the header file <math.h>. This macro will call a function which will either return $+\infty$ on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.
If the correct value overflows, *hypot* will return HUGE_VAL.

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in SVID.

## 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.

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

lockf — record locking on files

## SYNOPSIS

```
#include <unistd.h>

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

## DESCRIPTION

The *lockf* routine will allow sections of a file to be locked. *Lockf* 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 a 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:

```
F_ULOCK    0    /* Unlock a previously locked section */
F_LOCK     1    /* Lock a section for exclusive use */
F_TLOCK    2    /* Test and lock a section for exclusive use */
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,* or backwards for a negative *size* (the preceding byte, up to but not including the current offset). If *size* is zero (0) the section from the current offset through {FCHR_MAX} 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 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 [EDEADLK] 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 [EAGAIN] 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 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*(3C), or *fcntl*(2) 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) routine may be used to provide a timeout facility in applications which require this facility.

## ERRORS

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

[EBADF] *fildes* is not a valid file descriptor.

[EAGAIN] *function* is F_LOCK or F_TLOCK and the section is already locked by another process.

[EDEADLK] *function* is F_LOCK or F_TLOCK and a deadlock would occur. Also the *function* is either of the above or F_ULOCK and the number of entries in the system lock table would exceed {LOCK_MAX}.

## RETURN VALUE

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

## SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), open(2), read(2), write(2), unistd(5).

## APPLICATION USAGE

Record and file locking should not be used in combination with the *stdio*(3S) routines: *fopen*(3S), *fread*(3S), *fwrite*(3S), etc. Instead, the

more primitive, non-buffered routines, e.g. *open*(2) should be used. 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.

## FUTURE DIRECTIONS

Mandatory or enforcement-mode file and record locking will be added.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except for corrections/wording changes as follows:

a) SVID had F_UNLOCK instead of F_ULOCK in sixth paragraph of description.

b) The word ''file'' has been substituted for ''open'' in the [EBADF] definition.

c) In SVID the end of [EDEADLK] reads: ''... and there are not enough entries in the system lock table to honor the request''.

d) In [EAGAIN] the SVID erroneously refers to F_TEST instead of F_TLOCK.

## 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 variable from the user's environment.

## FILES

/etc/profile

## SEE ALSO

environ(5).

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

## NAME

lsearch, lfind — linear search and update

## SYNOPSIS

```
#include <search.h>

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

char *lfind (key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned *width;
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. *Width* is the size of an element in bytes. *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 (0) 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.

## RETURN VALUE

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.

## SEE ALSO

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

## APPLICATION USAGE

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.

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

## EXAMPLE

This fragment will read in <= TABSIZE strings of length <= 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);
        . . .
```

## FUTURE DIRECTIONS

A NULL pointer will be returned with *errno* set appropriately, if there is not enough room in the table to add a new item.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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.

*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.

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 zero (0).

| | |
|---|---|
| M_NLBLKS | Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than zero. 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 zero. 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 information describing space usage. It returns the structure *mallinfo* which includes the following members:

```
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.

## RETURN VALUE

*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.

## APPLICATION USAGE

This is the SVID version of *malloc*. An older (smaller) form may also exist, providing only the functionality of *malloc*, *free*, *realloc* and *calloc*. If both forms are present, it is the responsibility of the application developers to ensure that the appropriate version is linked into their applications. The system documentation will indicate in which library which version of *malloc* can be found.

## RELATIONSHIP TO SVID

Identical to the SVID *malloc*(OS) entry.

## NAME

matherr — error-handling function (OPTIONAL)

## 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 includes the following members:

```
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 mathematical 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.

## FUTURE DIRECTIONS

The mathematical functions which return HUGE or ±HUGE on overflow will return HUGE_VAL or ±HUGE_VAL respectively.

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 = 0) | M, —H | - | - | - | - | - |
| EXP: | - | - | H | 0 | - | - |
| LOG, LOG10: | | | | | | |
| (arg < 0) | M, —H | - | - | - | - | - |
| (arg = 0) | - | M, —H | - | - | - | - |
| POW: | - | - | ±H | 0 | - | - |
| neg ** non-int | M, 0 | - | - | - | - | - |
| 0 ** non-pos | | | | | | |
| SQRT: | M, 0 | - | - | - | - | - |
| GAMMA: | - | M, H | H | - | - | - |
| HYPOT: | - | - | H | - | - | - |
| SINH: | - | - | ±H | - | - | - |
| COSH: | - | - | H | - | - | - |
| SIN, COS, TAN: — | - | - | - | M, 0 | * | |
| ASIN, ACOS, ATAN2: 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. |

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in the SVID.

## NAME

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*.

## SEE ALSO

string(3C).

## APPLICATION USAGE

All these functions are declared in the *optional* <**memory.h**> header file.

*Memcmp* uses native character comparison. The sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

Character movement is performed differently in different implementations. Thus overlapping moves may be unpredictable.

## FUTURE DIRECTIONS

The declarations in <**memory.h**> will move to <**string.h**>.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 X's; *mktemp* will replace the X's with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

## APPLICATION USAGE

It is possible to run out of letters in which case a NULL pointer is returned.

## SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

## FUTURE DIRECTIONS

A NULL pointer will be returned if a unique name cannot be created.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

monitor — prepare execution profile (OPTIONAL)

## SYNOPSIS

```
#include <mon.h>

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

## DESCRIPTION

*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 zero (0) for this use of *monitor*. At most *nfunc* call counts can be kept.

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).

## SEE ALSO

profil(2), end(3C).

## RELATIONSHIP TO SVID

This *optional* function is not included in the SVID. It comes from UNIX System V Release 2.0.

## 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. 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 in this table to get the message string without the new-line. *Sys_nerr* is the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

## RELATIONSHIP TO SVID

Identical to the SVID entry. This is "level 2" in SVID, see SUBJECT TO CHANGE in Chapter 1. In the FUTURE DIRECTIONS section, the SVID states: "New error handling routines will be added to support the System V Error Message Standard as a tool for application developers to use".

## 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

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a command line and an I/O mode, either *"r"* for reading or *"w"* for writing. *Popen* creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer 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.

## SEE ALSO

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

## RETURN VALUE

*Popen* returns a NULL pointer if files or processes cannot be created, or if the command cannot be executed.

*Pclose* returns —1 if *stream* is not associated with a *popen* command.

## APPLICATION USAGE

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

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

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that in the SVID the first sentence of the RETURN VALUE section ends: ''...or if the shell cannot be accessed''.

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

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

> Zero or more *flags*, which modify the meaning of the conversion specification.
>
> An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '-', described below, has been given) to the field width.
>
> A *precision* that gives the minimum number of digits to appear for the *'d'*, *'o'*, *'u'*, *'x'*, or *'X'* conversions, the number of digits to appear after the decimal point for the *'e'* and *'f'* conversions, the maximum number of significant digits for the *'g'* conversion, or the maximum number of characters to be printed from a

string in *'s'* conversion. The precision takes the form of a period ('.') followed by a decimal digit string; a null digit string is treated as zero.

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

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

A field width or precision may be indicated by an asterisk ('*') instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

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

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

blank If the first character of a signed conversion is not a sign, a blank will be 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'*, *'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 as they normally are.

The conversion characters and their meanings are:

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

*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. However, if the value to be printed is greater than or equal to 1E+100, additional exponent digits will be pointed as necessary.

*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.

If the character after the *'%'* is not a valid conversion character, the results of the conversion are not predictable.

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.

## SEE ALSO

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

## FUTURE DIRECTIONS

*Printf* will make available character string representation for ∞ and "Not a Number" (NaN: a symbolic entity encoded in floating point format) to support the IEEE P754 standard.

## EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d",
        weekday, month, day, hour, min);
```

To print $\pi$ to 5 decimal places:

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

RELATIONSHIP TO SVID

Identical to the SVID entry, except for the removal of the part of the FUTURE DIRECTIONS section which in the SVID reads as follows:

> "System V currently allows leading zero padding to be specified by prepending a zero to the field width. The documentation of this feature will be removed to describe the preferred usage for the future application development. Ultimately this feature will be unsupported and customers will be warned of using an unsupported feature."

The SVID has in fact already incorporated this FUTURE DIRECTION.

## 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.

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*(3C) or *setvbuf*(3C) may be used to change the stream's buffering strategy.

## RETURN VALUE

On success, these functions each return the value they have written. 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 be grown. Because EOF is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

## SEE ALSO

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

## APPLICATION USAGE

Because it is implemented as a macro, *putc*(3S) treats incorrectly a *stream* argument with side effects. In particular, *putc(c, *f++);* doesn't work sensibly. *Fputc*(3S) should be used instead.

Because of possible differences in word length and byte ordering, files written using *putw*(3S) are machine-dependent, and may not be read using *getw*(3S) on a different processor.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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*.

## RETURN VALUE

*Putenv* returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise zero (0).

## SEE ALSO

exec(2), getenv(3C), malloc(3X).

## APPLICATION USAGE

*Putenv* manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*(3C). However, *envp* (the third argument to *main*) is not changed.
This routine uses *malloc*(3X) 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 return from the calling function while *string* is still part of the environment.

## RELATIONSHIP TO SVID

Identical to the SVID entry, except for the addition of the statement in the APPLICATIONS USAGE section: "After *putenv* is called, environment variables are not in alphabetical order".

## NAME

putpwent — write password file entry

## SYNOPSIS

```
#include <pwd.h>

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

## DESCRIPTION

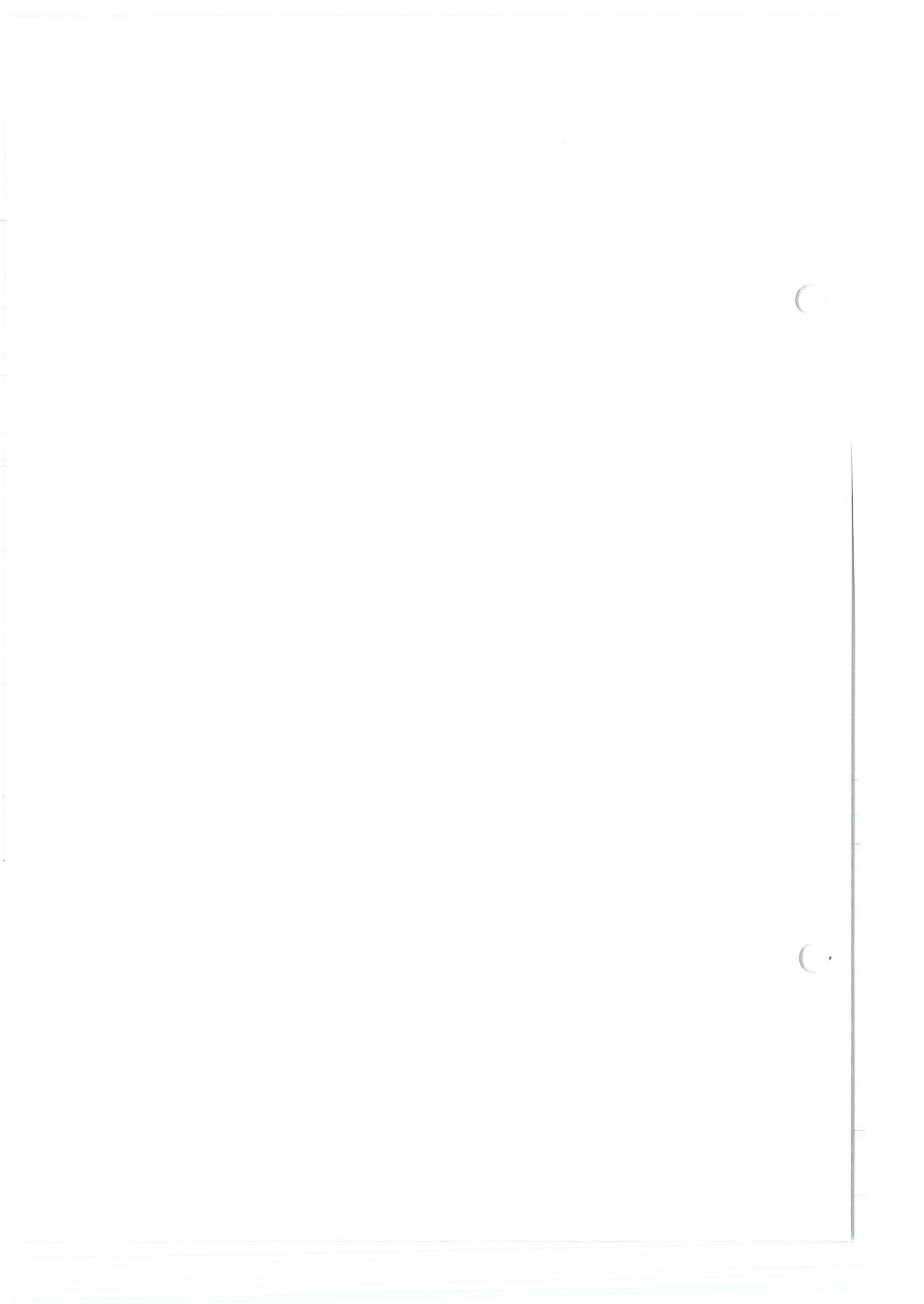
*Putpwent* is the inverse of *getpwent*(3C). Given a pointer to a *passwd* structure created by *getpwent*(3C) (or *getpwent*(3C) or *getpwnam*(3C)), *putpwent* writes a line on the stream *f* which matches the format of /etc/passwd.

## DIAGNOSTICS

*Putpwent* returns non-zero if an error was detected during its operation, otherwise zero (0).

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.



## 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.

## RETURN VALUE

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.

## SEE ALSO

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

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

qsort — quicker sort

## SYNOPSIS

```
void qsort (base, nel, width, compar)
char *base;
unsigned nel, width;
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. *Width* is the size of an element in bytes. *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.

## SEE ALSO

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

## APPLICATION USAGE

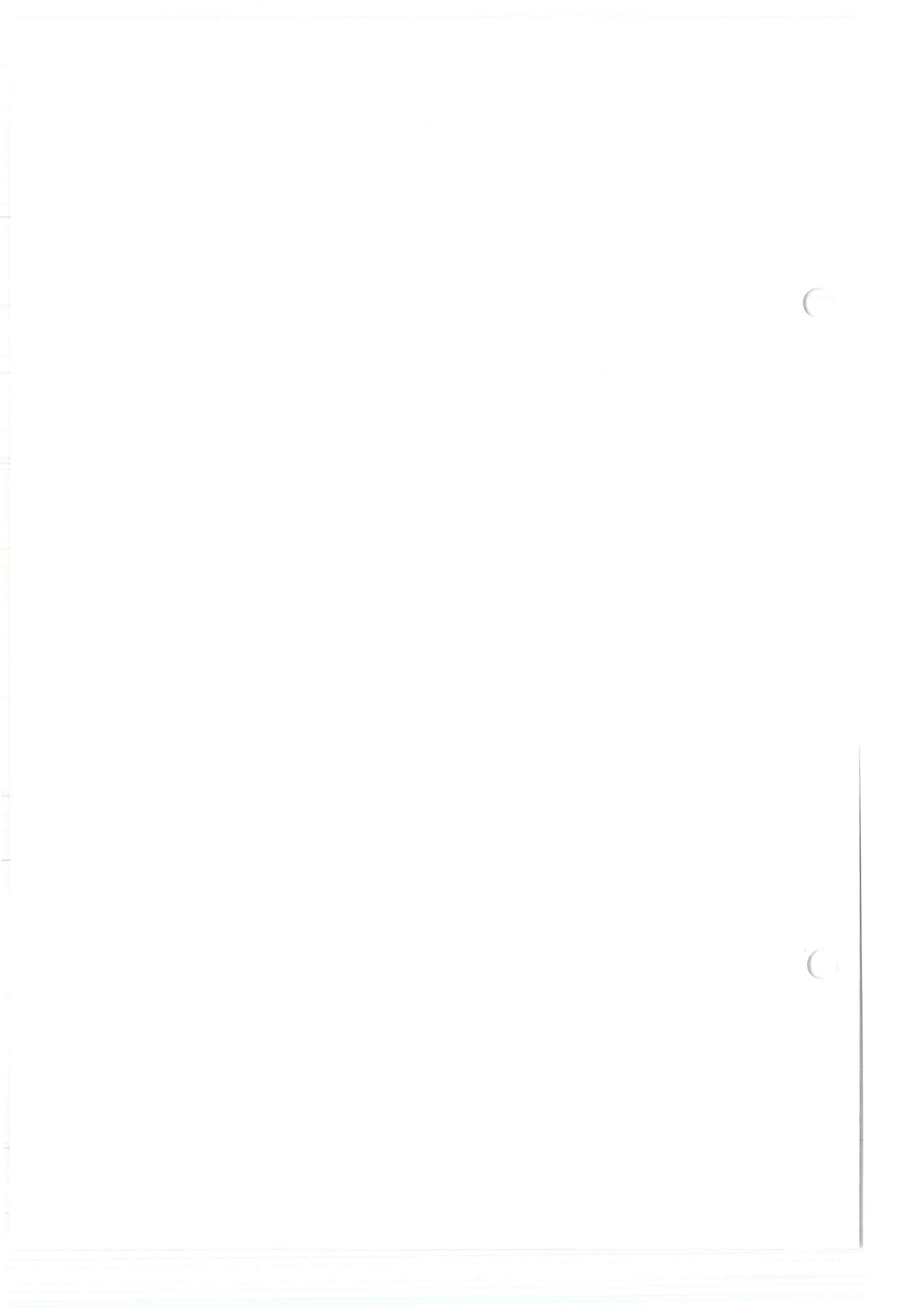
The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

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.

## RELATIONSHIP TO SVID

Identical to the SVID entry.



## 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 one (1).

## SEE ALSO

drand48(3C).

## APPLICATION USAGE

*Drand48*(3C) provides a much more elaborate random number generator.

The following functions define the semantics of *rand* and *srand.*

```
static unsigned long int next = 1;
int rand()
{
        next = next * 1103515245 + 12345;
        return((unsigned int)(next/65536) % 32768);
}
void srand(seed)
unsigned int seed;
{
        next = seed;
}
```

Specifying the semantic makes it possible to reproduce the behavior of programs that use pseudo-random sequences. This facilitates the testing of portable applications in different implementation.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 and returns a pointer to the compiled form. *Malloc*(3X) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument.

*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. The following are the valid symbols and their associated meanings.

. Matches any character execpt new-line.

* Matches *zero* or more occurences of the preceding character. The longest leftmost string that permits a match is chosen.

[] A non-empty string of characters in square brackets specifies a regular expression that matches *any one* character in that string.

- 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 -.

^ at the beginning of a regular expression matches an initial segment of string. A ^ immediatly following the left of a pair of square brackets ([]) causes the minus (-) to lose its special meaning within brackets, when it (-) occurs first or last in the string.

$ Matches the end of the string; *n* matches a new-line.

+ 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 to be used as themselves.

## SEE ALSO

malloc(3X).

## APPLICATION USAGE

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(3X) reuses the same vector saving time and space:

```
/* user's program */
        ...
char *
malloc(n)
unsigned n;
{
        static char rebuf[512];
        return (n <= sizeof rebuf) ? rebuf : NULL;
}
```

## EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
        ...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (ie. address of character '2' after the string "Testing3"). The string "Testing3" will be copied to the character array *ret0*.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 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.

*n* returns total number of characters so far which have been scanned from the beginning of the scan.

*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* is used for general integer conversion. it reads the input as an integer using C rules for conversion. for example, 12 would be read as 12( decimal), 012 would be read as 10 (decimal) and 0X12 as 18 (decimal). The corresponding argument should be an integer pointer.

*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.

If an invalid conversion character follows the *'%'*, the results of the operation may not be predictable.

The conversion characters *'d'*, *'u'*, *'o'*, and *'x'* 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 EO, 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.

## RETURN VALUE

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

## SEE ALSO

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

## FUTURE DIRECTIONS

*scanf* will make available character string representations for ∞ and "Not a Number" (NaN: a symbolic entity encoded in floating point format) to support the IEEE P754 standard.

## APPLICATION USAGE

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

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

## EXAMPLES

The call:

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

with input:

```
56789 0123 56a72
```

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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.

## RETURN VALUE

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

## SEE ALSO

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

## APPLICATION USAGE

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.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 zero (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* (the caller of 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. All accessible data have values as of the time *longjmp* was called.

## SEE ALSO

signal(2).

## APPLICATION USAGE

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.

If the call to *longjmp* is in a different function from the coresponding call to *setjmp*, register variables may have unpredictable values.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

sinh, cosh, tanh — hyperbolic functions (OPTIONAL)

## 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.

## RETURN VALUE

*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).

## SEE ALSO

matherr(3M).

## FUTURE DIRECTIONS

A macro HUGE_VAL will be defined in the header file <math.h>. This macro will call a function which will either return $+\infty$ on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.

*Sinh* and *cosh* will return HUGE_VAL (*sinh* will return —HUGE_VAL for negative *n*) when the correct value overflows.

## RELATIONSHIP TO SVID

Identical to the SVID entry. The *optional* mathematical group is mandatory in SVID.

## 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 wake-ups occur at fixed intervals, and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. 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).

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that "1-second" has been deleted from after "fixed" in the second sentence of the DESCRIPTION section.

## 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 made available to users for their own purposes.

Software signals made available to users are listed in *signal*(2). 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 signal listed in *signal*(2) 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 zero (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).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

stdio — standard buffered input/output package

## SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

## DESCRIPTION

The functions described as Standard I/O routines *(stdio)* 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 designates a nonexistent pointer.

An integer-constant EOF 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 Standard I/O related functions and constants 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*.

## RETURN VALUE

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

### SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(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(3C), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

### RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

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

## SYNOPSIS

```
#include <string.h>

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

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

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

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

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

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

int strlen (s)
char *s;

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

## SEE ALSO

memory(3C).

## APPLICATION USAGE

All these functions are declared in the *optional* <**string.h**> header file.

*Strcmp* and *strncmp* use native character comparison. The sign of the value returned when one of the characters has its high-order bit set is implementation-dependent.

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

## FUTURE DIRECTIONS

The type of the argument *n* to *strncat, strncmp* and *strncpy* and the type of the value returned by *strlen* will be declared through the **typedef** facility in a header file as *size_t*.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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, followed by an integer.

If the value of *ptr* is not *(char **)0*, 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 **)0)*.

## RETURN VALUE

If the correct value would cause overflow, ±HUGE 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].

## SEE ALSO

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

## FUTURE DIRECTIONS

A macro HUGE_VAL will be defined in the header file <math.h>. This macro will call a function which will either return $+\infty$ on a system supporting IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.
If the correct value overflows, *strtod* will return ±HUGE_VAL (according to the sign of the value).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 **)0*, 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 **)0, 10)*.

*Atoi(str)* is equivalent to *(int) strtol(str, (char **)0, 10)*.

## SEE ALSO

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

## APPLICATION USAGE

Overflow conditions are ignored.

## FUTURE DIRECTIONS

Error handling will be added to *strtol.*

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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.

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

system — issue a command

## SYNOPSIS

```
#include <stdio.h>

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

## DESCRIPTION

*System* causes the *string* to be given to a command interpreter and execution process as input.

### Definitions

A *blank* is a tab or a space. A *parameter name* is a sequence of letters, digits, or underscores beginning with a letter or underscore. A *parameter* is a parameter name, a digit, or any of the characters *, @, #, ?, —, $, and !.

### Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the pathname or file name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0, see *exec*(2). The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see *signal*(2) for a list of status values).

A *pipeline* is a sequence of two or more *commands* separated by |. The standard output of each command but the last is connected by a *pipe*(2) to the standard input of the next command. Each command is run as a separate process; the command execution process waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more simple commands or pipelines separated by ;, &, &&, or | |. A list may optionally be terminated by ; or &. Of these four symbols, ; and & have equal precedence, which is lower than that of && and | |. The symbols && and | | also have equal precedence.

> A semicolon (;) causes sequential execution of the preceding pipeline;
>
> An ampersand (&) causes asynchronous execution of the preceding pipeline;
>
> The symbol && ( | | ) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrarily long sequence of new-lines may appear in

a *list*, instead of a semicolon, to delimit commands.

A *command* is either a *simple-command* or one of the following. Unless otherwise stated, the value returned by a command is that of the last *simple-command* executed in the command list.

### Comments

A word beginning with # causes that word and all the following characters up to a new-line to be ignored.

### Command Substitution

The standard output from a command enclosed in a pair of grave accents (`) may be used as part or all of a word; trailing new-lines are removed.

### Parameter Substitution

The character **$** is used to introduce substitutable *parameters*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Keyword parameters (also known as variables) may be assigned values by writing:

*parameter-name=value*

*parameter-name=value*

....

${*parameter*}
The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is * or @, all the positional parameters, starting with **$1**, are substituted (separated by spaces). Parameter **$0** is set from argument zero.

The following parameters are automatically set:

- \# The number of positional parameters in decimal.
- ? The decimal value returned by the last synchronously executed command.
- $ The process number of this process.
- ! The process number of the last background command invoked.

The following parameters are used by the command execution process:

- HOME The initial working (home) directory, initially set from the 6th field in the **/etc/passwd** file (see Appendix 2.9).
- PATH The search path for commands (see *Execution* below).

### Blank Interpretation

After parameter and command substitution, the results of substitution are

scanned for internal field separator characters (space, tab and newline) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

### File Name Generation

Following substitution, each command *word* is scanned for the characters *, ?, and [. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character . at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly.

| | |
|---|---|
| * | Matches any string, including the null string. |
| ? | Matches any single character. |
| [...] | Matches any one of the enclosed characters. A pair of characters separated by — matches any character lexically between the pair, inclusive. If the first character following the opening '[' is a "!" any character not enclosed is matched. |

### Quoting

The following characters have a special meaning and cause termination of a word unless quoted:

; & ( ) | ^ < > new-line space tab

A character may be *quoted* (i.e., made to stand for itself) by preceding it with a \. The pair \new-line is ignored. All characters enclosed between a pair of single quote marks (''), except a single quote, are quoted. Inside double quote marks (""), parameter and command substitution occurs and \ quotes the characters \, `, ", and $. "$*" is equivalent to "$1 $2 ...", whereas "$@" is equivalent to "$1" "$2" ....

### Input/Output

Before a command is executed, its input and output may be redirected using a special notation. The following may appear anywhere in a *simple-command* or may precede or follow a *command* and are *not* passed on to the invoked command; substitution occurs before *word* or *digit* is used:

| | |
|---|---|
| <word | Use file *word* as standard input (file descriptor 0). |
| >word | Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length. |
| >>word | Use file *word* as standard output. If the file exists output is appended to it (by first seeking to the end-of-file); |

otherwise, the file is created.

<<[-]word — The shell input is read up to a line that is the same as *word*, or to an end-of-file. The resulting document becomes the standard input. If any character of *word* is quoted, no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, (unescaped) **new-line** is ignored, and \ must be used to quote the characters \, $, `, and the first character of *word*. If — is appended to <<, all leading tabs are stripped from *word* and from the document.

<&digit — Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using >**&digit**.

<&- — The standard input is closed. Similarly for the standard output using >&-.

If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

```
... 1>xxx 2>&1
```

first associates file descriptor 1 with file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e. *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

If a command is followed by & the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking process as modified by input/output specifications.

### Environment

The *environment* (see *exec*(2)) is a list of parameter name-value pairs that is passed to an executed program in the same way as a normal argument list. On invocation, the environment is scanned and a parameter is created for each name found, giving it the corresponding value.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters.

TERM=450 cmd;

### Signals

The SIGINT and SIGQUIT signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the command execution process from its parent.

### Execution

Each time a command is executed, the above substitutions are carried out. A new process is created and an attempt is made to execute the command via *exec*(2).

The parameter PATH defines the search path for the directory containing the command. Alternative directory names are separated by a colon (:). The default path is :/bin:/usr/bin (specifying the current directory, /bin, and /usr/bin, in that order). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a / the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an *a.out* file, it is assumed to be a file containing commands.

Conventionally, *system* has been implemented with the Bourne shell, *sh*. The current definition of *system* is not intended to preclude that or its implementation by another command interpreter that provides the minimum functionality described here. Of course, any implementation may provide a superset of the functionality described.

## RETURN VALUE

Errors, such as syntax errors, cause a non-zero return value and execution of the command file is abandoned. Otherwise, the exit status of the last command executed is returned.

## FILES

/dev/null

## SEE ALSO

dup(2), exec(2), fork(2), pipe(2), signal(2), ulimit(2), umask(2), wait(2).

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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 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+*").

## RETURN VALUE

If the file cannot be opened, an error message is printed and a NULL pointer is returned.

## SEE ALSO

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

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that the RETURN VALUE sentence has also been repeated in the DESCRIPTION section.

## 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 which 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.

*Tempnam* uses *malloc*(3X) 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*(3X)). If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3X) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

## SEE ALSO

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

## APPLICATION USAGE

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.

The functions *tmpnam* and *tempnam* 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 *fclose*(2) or *unlink*(2) to remove the file when its use is ended.

If called more than {TMP_MAX} 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 so as to render duplication by other means unlikely.

### RELATIONSHIP TO SVID

Identical to the SVID entry.

## NAME

sin, cos, tan, asin, acos, atan, atan2 — trigonometric functions (OPTIONAL)

## 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 $-\pi/2$ to $\pi/2$.

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

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

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

## RETURN VALUE

*Sin*, *cos*, and *tan* lose accuracy when their argument is far from zero (0). 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).

## SEE ALSO

matherr(3M).

## RELATIONSHIP TO SVID

Identical to SVID entry. The *optional* mathematical group is mandatory in SVID.

## NAME

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

## SYNOPSIS

```
#include <search.h>

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

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

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

void twalk (root, action)
char *root;
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 zero (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.

## RETURN VALUE

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 *tsearch, 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.

## SEE ALSO

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

## APPLICATION USAGE

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.

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.

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

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, &root,
                        node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk(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)
struct node **node;
VISIT order;
int level;
{
        if (order == preorder || order == leaf) {
                (void)printf("string = %20s, length = %d\n",
                        (*node)->string, (*node)->length);
        }
}
```

## RELATIONSHIP TO SVID

Identical to the SVID entry, except that a programming error in the *node_compare* part of the example has been corrected. The SVID reads:

```
        return strcmp(node1—>string, node2—>string);
```

## 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, zero (0) otherwise.

## FILES

/dev/*

## RETURN VALUE

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

## APPLICATION USAGE

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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.

## FILES

/etc/utmp

## SEE ALSO

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

## DIAGNOSTICS

A value of zero (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.

## RELATIONSHIP TO SVID

This function is not included in the SVID. It comes from UNIX System V Release 2.0.

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

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

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

## RETURN VALUE

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

## SEE ALSO

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

## RELATIONSHIP TO SVID

Identical to the SVID entry.

## 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.h>.

## SEE ALSO

printf(3S).

## EXAMPLE

The following demonstrates how *vfprintf* could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *              error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *       separately declared because of the definition of varargs.
 */
```

```
va_dcl
{
        va_list args;
        char *fmt;

        va_start(args);
        /* print out name of function causing error */
        (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
        fmt = va_arg(args, char *);
        /* print out remainder of message */
        (void)vfprintf(fmt, args);
        va_end(args);
        (void)abort( );
}
```

## RELATIONSHIP TO SVID

Identical to the SVID entry.