NAME

intro – introduction to subroutines and libraries

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

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

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

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

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

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

Two groups of entries represent direct communication with RTOS. Functions whose names begin with **of** ("outside file system") provide RTOS-style input/output. Functions whose names begin with **qu** ("queue") provide access to RTOS queue management.

DEFINITIONS

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

FILES

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

SEE ALSO

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

DIAGNOSTICS

Functions in the Math Library (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* (see *intro*(2)) is set to the value EDOM or ERANGE.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in section 2 (*System Calls*). If a program inadvertantly defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for sections 2, 3C, and 3S

are checked automatically. Other definitions can be included by using the **-l** option (for example, **-l***m* includes definitions for the Math Library, section 3M). Use of *lint* is highly recommended.

NAME

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

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long l;

DESCRIPTION

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

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

A64l takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l* will use the first six.

L64a takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, *l64a* returns a pointer to a null string.

BUGS

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

NAME

abort – generate an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

Abort first closes all open files if possible, then causes an IOT signal to be sent to the process. This usually results in termination with a core dump.

It is possible for *abort* to return control if **SIGIOT** is caught or ignored, in which case the value returned is that of the *kill*(2) system call.

SEE ALSO

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

DIAGNOSTICS

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

NAME

abs – return integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

Abs returns the absolute value of its integer operand.

BUGS

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

SEE ALSO

floor(3M).

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
int expression;

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), *assert* prints

"Assertion failed: *expression*, file *xyz*, line *nnn*"

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the *assert* statement.

Compiling with the preprocessor option **–DNDEBUG** (see *cpp* (1)), or with the preprocessor control statement "**#define** NDEBUG" ahead of the "**#include** <assert.h>" statement, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

NAME

atof – convert ASCII string to floating-point number

SYNOPSIS

double atof (nptr)
char *nptr;

DESCRIPTION

Atof converts a character string pointed to by *nptr* to a double-precision floating-point number. The first unrecognized character ends the conversion. *Atof* recognizes an optional string of white-space characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional **e** or **E** followed by an optionally signed integer. If the string begins with an unrecognized character, *atof* returns the value zero.

DIAGNOSTICS

When the correct value would overflow, *atof* returns **HUGE**, and sets *errno* to **ERANGE**. Zero is returned on underflow.

SEE ALSO

scanf(3S).

NAME

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

SYNOPSIS

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
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.

DIAGNOSTICS

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

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

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

SEE ALSO

matherr(3M).

NAME

bsearch – binary search a sorted table

SYNOPSIS

#include <search.h>

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

DESCRIPTION

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

EXAMPLE

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

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

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

#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)
struct node *node1, *node2;
{
        return strcmp(node1->string, node2->string);
}
```

NOTES

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

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).

The resolution of the clock is 16.667 milliseconds on operating system Processors.

SEE ALSO

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

BUGS

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

NAME

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

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

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

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

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

SEE ALSO

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

NAME

crypt, setkey, encrypt – generate 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* entries provide (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 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 or decrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

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

BUGS

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

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>

char *ctermid(s)
char *s;

DESCRIPTION

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

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

NOTES

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

SEE ALSO

ttyname(3C).

NAME

csinit - initialize a character-set translation table

SYNOPSIS

```
#define  CSMAXSIZ       1
#include <cs.h>
#include <ctype.h>
#include <stdio.h>

struct csttbl *
csinit (filename, silent, status)
register char *filename;
register int silent;
register int *status;
```

DESCRIPTION

Csinit(3X) constructs a character-set translation data structure from a character-set translation source file. Csinit reads the source file named by its <filename> argument, converts it to a csttbl character-set translation table, and returns a pointer to that structure. Validation of the character-set translation source file is performed. The RETURN value of csinit is NULL if the conversion operation was unsuccessful. The csttbl structure is shown in Figure csinit-1.

```
/*
 * Character set translation table argument.
 * The user program should define CSMAXSIZ as the maximum translation
 * table size it is prepared to handle and set cs_tmax to that value.
 */
#ifdef CSMAXSIZ
struct  csttbl  {
        int             cs_tmax;        /* should be set to CSMAXSIZ */
        union   {
                struct cstthdr  cs_hdr;
                char            cs_tbl[CSMAXSIZ];
        }cs_u;
};
#endif
```

Figure Csinit-1. Csttbl Structure

Csinit arguments are:

<filename> Name of the character-set translation source file.

<silent> Flag to select or deselect printing of error messages. If the <silent> argument is FALSE, then diagnostics are written on the standard error file.

<status> Status word to reflect completion status. Values for completion status are defined in the cs.h header file.

This routine resides in the file /usr/lib/libcs.a. The program must be loaded with the object-file, access-routine, library libcs.a.

DIAGNOSTICS

When the <silent> argument is FALSE, csinit writes error messages of the following form on its standard error file. The %d represents the line number of the translation table at which the error occurred; %n represents the character-set number.

line %d - redeclaration of character set %n
line %d - undefined character set number %n
line %d - format7 statement unexpected
line %d - inbound statement unexpected
line %d - outbound statement unexpected
line %d - number of entries does not match defined range
line %d - translate statement missing accent value
line %d - translate statement missing character set number
line %d - translate statement missing high range value
line %d - translate statement missing input sequence
line %d - translate statement missing low range value
line %d - no primary character set defined
line %d - translate statement missing range keyword
line %d - syntax error

SEE ALSO

cstrans(3X), cstermio(7)

NAME

cstrans - perform character-set translation

SYNOPSIS

```
#include <sys/csintern.h>
#include <cs.h>

cstrans (csdp)
    register CSDATP     csdp;
```

DESCRIPTION

Cstrans translates characters from one buffer to another through a translation table. It translates characters until either the output buffer becomes full or the input buffer is empty.

Its argument, <csdp>, is the address of a data structure that points to an input buffer, a translation table, and an output buffer, and contains information describing the current state of the translation.

This subroutine package handles translation of data that may be represented as XSIS 058404 strings, external device codes, or internal 16-bit characters. Input data is in a buffer of unsigned char or short. The output data is placed into a similar buffer. For outbound characters that are not in internal character-set 0 and that have no declared entry in the translation table, cstrans(3Y) substitutes a question mark character (?).

There are five translation modes, all of which use an internal 16-bit character input or output buffer, with the other buffer being either 8-bit characters or internal 16-bit characters. Table cstrans-1 describes these modes.

Table Cstrans-1. Character Translation Modes

Mode	Function
0	Translate from internal 16-bit to internal 16-bit using an internal translation table. This mode either enforces the Motorola private character-set or avoids the character sets for Motorola private, ligature, and accented characters.
1	Translate from external-device character code to internal 16-bit characters through an external-device translation table.

Table Cstrans-2. Character Translation Modes (Continued)

Mode	Function
2	Translate from internal 16-bit characters to XSIS 058404 strings with options for 16-bit stringlets or for 7-bit representations.
3	Translate from XSIS 058404 strings to internal 16-bit characters.
4	Translate from internal 16-bit characters to external-device character codes through an external-device translation table.

As an output filter, three translations would be applied in sequence:

Mode 2	Mode 0 using cs_tostd	Mode 4 using a device-specific translation table

To reformat XSIS 058404 strings, four translations could be applied. For example, to reformat them to Motorola, private, character-set 040 strings, use the following sequence:

Mode 2	Mode 0 using cs_tostd	Mode 0 using cs_topri	Mode 2

Other combinations of translation modes can be used. The only requirement is that each output buffer must be in the form expected for the next translation's input buffer.

The external, device-translation input sections must provide characters in the standard internal character sets, avoiding sets 040, 360, and 361. They may assume that their input comes from that same standard form. The cs_tostd translation table is applied to input strings to ensure the standard input form.

This convention means that output translation tables do not have to handle all the different forms that are legal. For example, the Ä dieresis symbol ( ) can be represented in three different internal forms:

<000><310> <000><101>	standard form: dieresis and "A"
<361><047>	the accented character rendering
<040><241>	the Motorola private form

Also, for devices that accept the ISO forms, no translation is required. For some hardcopy devices that don't accept the ISO form, the accents can still be mapped to <accent> and <backspace>.

This routine resides in the file /usr/lib/libcs.a. The program must be loaded with the object-file, access-routine library libcs.a.

SEE ALSO

csinit(3X)

Series 6000 International Support Package Reference Manual

NAME

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

SYNOPSIS

```
#include <time.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. All the fields have constant width.

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

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

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

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

```
struct tm {
        int tm_sec; /* seconds (0 - 59) */
        int tm_min; /* minutes (0 - 59) */
        int tm_hour; /* hours (0 - 23) */
        int tm_mday; /* day of month (1 - 31) */
        int tm_mon; /* month of year (0 - 11) */
        int tm_year; /* year - 1900 */
        int tm_wday; /* day of week (Sunday = 0) */
        int tm_yday; /* day of year (0 - 365) */
        int tm_isdst;
};
```

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

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

If an environment variable named **TZ** is present, *asctime* uses the contents of the variable to override the default time zone. The value of **TZ** must be a three-letter time zone name, followed

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

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

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

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

SEE ALSO

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

BUGS

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

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

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, 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 less than 0200.

DIAGNOSTICS

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

SEE ALSO

ascii(5).

NAME

curses – CRT screen handling and optimization package

SYNOPSIS

#include <curses.h>
cc [flags] files **–lcurses** [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. In order to initialize the routines, the routine *initscr( )* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin( )* should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs want this) after calling *initscr( )* you should call *"nonl( ); cbreak( ); noecho( );"*

The full curses interface permits manipulation of data structures called *windows* which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called **stdscr** is supplied, and others can be created with **newwin**. Windows are referred to by variables declared "WINDOW *", the type WINDOW is defined in curses.h to be a C structure. These data structures are manipulated with functions described below, among which the most basic are **move**, and **addch**. (More general versions of these functions are included with names beginning with 'w', allowing you to specify a window. The routines not beginning with 'w' affect **stdscr**.) Then *refresh( )* is called, telling the routines to make the users CRT screen look like **stdscr**.

Mini-Curses is a subset of curses which does not allow manipulation of more than one window. To invoke this subset, use –DMINICURSES as a **cc** option. This level is smaller and faster than full curses.

If the environment variable TERMINFO is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is **/usr/lib/terminfo**, and TERM is set to "vt100", then normally the compiled file is found in **/usr/lib/terminfo/v/vt100**. (The "v" is copied from the first letter of "vt100" to avoid creation of huge directories.) However, if TERMINFO is set to **/usr/mark/myterms**, curses will first check **/opusr/mark/myterms/v/vt100**, and if that fails, will then check **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

SEE ALSO

terminfo(4).

FUNCTIONS

Routines listed here may be called when using the full curses. Those marked with an asterisk may be called when using Mini-Curses.

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
box(win, vert, hor)	draw a box around edges of *win* *vert* and *hor* are chars to use for vert. and hor. edges of box
clear( )	clear *stdscr*
clearok(win, bf)	clear screen before next redraw of *win*

clrtobot( )	clear to bottom of *stdscr*
clrtoeol( )	clear to end of line on *stdscr*
cbreak( )*	set cbreak mode
delay_output(ms)*	insert ms millisecond pause in output
delch( )	delete a character
deleteln( )	delete a line
delwin(win)	delete *win*
doupdate( )	update screen from all wnooutrefresh
echo( )*	set echo mode
endwin( )*	end window modes
erase( )	erase *stdscr*
erasechar( )	return user's erase character
fixterm( )	restore tty to "in curses" state
flash( )	flash screen or beep
flushinp( )*	throw away any typeahead
getch( )*	get a char from tty
getstr(str)	get a string through *stdscr*
gettmode( )	establish current tty modes
getyx(win, y, x)	get (y, x) co-ordinates
has_ic( )	true if terminal can do insert character
has_il( )	true if terminal can do insert line
idlok(win, bf)*	use terminal's insert/delete line if bf != 0
inch( )	get char at current (y, x) co-ordinates
initscr( )*	initialize screens
insch(c)	insert a char
insertln( )	insert a line
intrflush(win, bf)	interrupts flush output if bf is TRUE
keypad(win, bf)	enable keypad input
killchar( )	return current user's kill character
leaveok(win, flag)	OK to leave cursor anywhere after refresh if flag!=0 for *win*, otherwise cursor must be left at current position.
longname( )	return verbose name of terminal
meta(win, flag)*	allow meta characters on input if flag != 0
move(y, x)*	move to (y, x) on *stdscr*
mvaddch(y, x, ch)	move(y, x) then addch(ch)
mvaddstr(y, x, str)	similar...
mvcur(oldrow, oldcol, newrow, newcol)	low level cursor motion
mvdelch(y, x)	like delch, but move(y, x) first
mvgetch(y, x)	etc.
mvgetstr(y, x)	
mvinch(y, x)	
mvinsch(y, x, c)	
mvprintw(y, x, fmt, args)	
mvscanw(y, x, fmt, args)	
mvwaddch(win, y, x, ch)	
mvwaddstr(win, y, x, str)	
mvwdelch(win, y, x)	
mvwgetch(win, y, x)	
mvwgetstr(win, y, x)	
mvwin(win, by, bx)	
mvwinch(win, y, x)	

mvwinsch(win, y, x, c)
mvwprintw(win, y, x, fmt, args)
mvwscanw(win, y, x, fmt, args)
newpad(nlines, ncols) — create a new pad with given dimensions
newterm(type, fd) — set up new terminal of given type to output on fd
newwin(lines, cols, begin_y, begin_x)
— create a new window
nl( )* — set newline mapping
nocbreak( )* — unset cbreak mode
nodelay(win, bf) — enable nodelay input mode through getch
noecho( )* — unset echo mode
nonl( )* — unset newline mapping
noraw( )* — unset raw mode
overlay(win1, win2) — overlay win1 on win2
overwrite(win1, win2) — overwrite win1 on top of win2
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
— like prefresh but with no output until doupdate called
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
— refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(fmt, arg1, arg2, ...)
— printf on *stdscr*
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
scanw(fmt, arg1, arg2, ...)
— scanf through *stdscr*
scroll(win) — scroll *win* one line
scrollok(win, flag) — allow terminal to scroll if flag != 0
set_term(new) — now talk to terminal new
setscrreg(t, b) — set user scrolling region to lines t through b
setterm(type) — establish terminal with given type
setupterm(term, filenum, errret)
standend( )* — clear standout mode attribute
standout( )* — set standout mode attribute
subwin(win, lines, cols, begin_y, begin_x)
— create a subwindow
touchwin(win) — change all of *win*
traceoff( ) — turn off debugging trace output
traceon( ) — turn on debugging trace output
typeahead(fd) — use file descriptor fd to check typeahead
unctrl(ch)* — printable version of *ch*
waddch(win, ch) — add char to *win*
waddstr(win, str) — add string to *win*
wattroff(win, attrs) — turn off *attrs* in *win*
wattron(win, attrs) — turn on *attrs* in *win*
wattrset(win, attrs) — set attrs in *win* to *attrs*

wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
wdelch(win, c)	delete char from *win*
wdeleteln(win)	delete line from *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win, str)	get a string through *win*
winch(win)	get char at current (y, x) in *win*
winsch(win, c)	insert char into *win*
winsertln(win)	insert line into *win*
wmove(win, y, x)	set current (y, x) co-ordinates on *win*
wnoutrefresh(win)	refresh but no screen output
wprintw(win, fmt, arg1, arg2, ...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win, fmt, arg1, arg2, ...)	scanf through *win*
wsetscrreg(win, t, b)	set scrolling region of *win*
wstandend(win)	clear standout attribute in *win*
wstandout(win)	set standout attribute in *win*

TERMINFO LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, *setupterm* should be called. This will define the set of terminal dependent variables defined in terminfo(4). The include files <curses.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parmeterized strings should be passed through *tparm* to instantiate them. All terminfo strings (including the output of tparm) should be printed with *tputs* or *putp* . Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

fixterm( )	restore tty modes for terminfo use (called by setupterm)
resetterm( )	reset tty modes to state before program entry
setupterm(term, fd, rc)	read in database. Terminal type is the character string *term*, all output is to operating system file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be *setupterm(0, 1, 0)* which uses all defaults.
tparm(str, p1, p2, ..., p9)	instantiate string str with parms p_i.
tputs(str, affcnt, putc)	apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *Putc* is a putchar-like function to which the characters are passed, one at a time.
putp(str)	handy function that calls tputs (str, 1, putchar)
vidputs(attrs, putc)	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar-like

	function *putc*.
vidattr(attrs)	Like vidputs but outputs through putchar

TERMCAP COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use termcap. Their parameters are the same as for termcap. They are emulated using the *terminfo* database. They may go away at a later date.

tgetent(bp, name)	look up termcap entry for name
tgetflag(id)	get boolean entry for id
tgetnum(id)	get numeric entry for id
tgetstr(id, area)	get string entry for id
tgoto(cap, col, row)	apply parms to given cap
tputs(cap, affcnt, fn)	apply padding to cap calling fn as putchar

ATTRIBUTES

The following video attributes can be passed to the functions *attron*,*attroff*,*attrset*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

FUNCTION KEYS

The following function keys might be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported, due to lack of definitions in *terminfo* or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab

KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNING

The plotting library *plot*(3X) and the curses library *curses*(3X) both use the names erase( ) and move( ). The curses versions are macros. If you need both libraries, put the *plot*(3X) code in a different source file than the *curses*(3X) code, and/or #undef move( ) and erase( ) in the *plot*(3X) code.

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

Cuserid gets the user's login name as found in **/etc/utmp**. If the login name cannot be found, *cuserid* gets the login name corresponding to the user ID of the process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least **L_cuserid** characters; the representation is left in this array. The constant **L_cuserid** is defined in the **<stdio.h>** header file.

DIAGNOSTICS

If the login name cannot be found and the process's owner lacks a password file entry, *cuserid* returns a NULL pointer; if *s* is not a NULL pointer, a null character **(\0)** will be placed at *s[0]*.

SEE ALSO

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

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
int fd;
```

DESCRIPTION

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

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

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

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

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

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

The *telno* element is for a pointer to a character string representing the telephone number to be dialed. Numbers consist of the following symbols:

0-9	dial 0-9
* or :	dial *
# or ;	dial #
–	4-second delay for second dial tone
e or <	end-of-number
w or =	wait for secondary dial tone
f	flash off hook for 1 second

On a smart modem, these symbols are translated to modem commands using the modem description in **/usr/lib/uucp/modemcap**.

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

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

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

FILES

/usr/lib/uucp/modemcap
/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK..*tty-device*

SEE ALSO

uucp(1C), alarm(2), read(2), write(2) modemcap(5), termio(7).

DIAGNOSTICS

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

```
INTRPT    -1    /* interrupt occurred */
D_HUNG    -2    /* dialer hung (no return from write) */
NO_ANS    -3    /* no answer within 10 seconds */
ILL_BD    -4    /* illegal baud-rate */
A_PROB    -5    /* acu problem (open() failure) */
L_PROB    -6    /* line problem (open() failure) */
NO_Ldv    -7    /* can't open LDEVS file */
DV_NT_A   -8    /* requested device not available */
DV_NT_K   -9    /* requested device not known */
NO_BD_A   -10   /* no device available at requested baud */
NO_BD_K   -11   /* no device known at requested baud */
```

WARNINGS

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

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

BUGS

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

NAME

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

SYNOPSIS

double drand48 ( )

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

long lrand48 ( )

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

long mrand48 ( )

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

void srand48 (seedval)
long seedval;

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

$a = 5\mathrm{DEECE66D}_{16} = 273673163155_8$
$c = \mathrm{B}_{16} = 13_8$.

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

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

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

NOTES

The versions of these routines for the VAX-11 and PDP-11 are coded in assembly language for maximum speed. It requires approximately 80 μsec on a VAX-11/780 and 130 μsec on a PDP-11/70 to generate one pseudo-random number. On other computers, the routines are coded in portable C. The source code for the portable version can even be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* do not exist; instead, they are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

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

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

SEE ALSO

rand(3C).

NAME

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

SYNOPSIS

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

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

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

DESCRIPTION

Ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

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

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

SEE ALSO

printf(3S).

BUGS

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

NAME

end, etext, edata – last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

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

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

SEE ALSO

brk(2), malloc(3C).

NAME

erf, erfc – error function and complementary error function

SYNOPSIS

#include <math.h>

double erf (x)
double x;

double erfc (x)
double x;

DESCRIPTION

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

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

SEE ALSO

exp(3M).

NAME

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

SYNOPSIS

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

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

double sqrt (x)
double x;
```

DESCRIPTION

Exp returns e^x.

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

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

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

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

DIAGNOSTICS

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

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

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

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

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

SEE ALSO

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

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.

DIAGNOSTICS

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

SEE ALSO

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

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

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

DESCRIPTION

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

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

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

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

NOTE

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

SEE ALSO

open(2), fopen(3S).

NAME

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

SYNOPSIS

#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

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

double fabs (x)
double x;

DESCRIPTION

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

Ceil returns the smallest integer not less than x.

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

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

SEE ALSO

abs(3C).

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

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

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

SEE ALSO

open(2), fclose(3S).

DIAGNOSTICS

Fopen and *freopen* return a NULL pointer on failure.

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>

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

int fwrite (ptr, size, nitems, stream)
char *ptr;
int size, 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*.

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

SEE ALSO

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

DIAGNOSTICS

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*.

NAME

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

SYNOPSIS

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

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

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

DESCRIPTION

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

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

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

DIAGNOSTICS

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

NAME

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

SYNOPSIS

```
#include <stdio.h>

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

void rewind (stream)
FILE *stream;

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

lseek(2), fopen(3S).

DIAGNOSTICS

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

WARNING

On this operating system and other systems derived from the UNIX System, the value returned by *ftell* is a number of bytes, and a program can use this value to seek relative to the current offset. Such programs are not portable to systems where file offsets are not measured in bytes.

NAME

ftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

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

DESCRIPTION

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

Ftw visits a directory before visiting any of its descendants.

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

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

SEE ALSO

stat(2), malloc(3C).

BUGS

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

It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.

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

NAME

gamma – log gamma function

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 Γ:

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

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

DIAGNOSTICS

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

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

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

SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

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

DESCRIPTION

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

WARNING

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

BUGS

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.

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

NAME

getcwd – get path-name of current working directory

SYNOPSIS

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

DESCRIPTION

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

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

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

EXAMPLE

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(1);
}
printf("%s\n", cwd);
```

SEE ALSO

pwd(1), malloc(3C), popen(3S).

DIAGNOSTICS

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

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (name)
char *name;

DESCRIPTION

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

SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

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

DESCRIPTION

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

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

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

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

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

FILES

/etc/group

SEE ALSO

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

DIAGNOSTICS

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

WARNING

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

BUGS

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

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

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

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

FILES

/etc/utmp

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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

NAME

getopt – get option letter from argument vector

SYNOPSIS

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

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

DESCRIPTION

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *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. Because *optind* is external, it is normally initialized to zero automatically 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.

DIAGNOSTICS

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 a non-zero value.

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;
        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)) {
        .
        .
        .
}
```

SEE ALSO

getopt(1).

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
char *prompt;

DESCRIPTION

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

FILES

/dev/tty

SEE ALSO

crypt(3C).

WARNING

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

BUGS

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

NAME

getpw – get name from UID

SYNOPSIS

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

DESCRIPTION

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

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

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

Getpw returns non-zero on error.

WARNING

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

NAME

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

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

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

DESCRIPTION

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

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

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

The *pw_comment* field is unused; the others have meanings described in *passwd*(4).

Getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *Getpwuid* searches from the beginning of the file until a numerical user id matching *uid* is found and returns a pointer to the particular structure in which it was found. *Getpwnam* searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

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

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

FILES

/etc/passwd

SEE ALSO

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

DIAGNOSTICS

A NULL pointer is returned on EOF or error.

WARNING

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

BUGS

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

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

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

DESCRIPTION

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

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

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

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

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

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

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

DESCRIPTION

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

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

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

Getutid searches forward from the current point in the *utmp* file until it finds an entry with a *ut_type* matching *id->ut_type* if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME or NEW_TIME. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS or DEAD_PROCESS, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

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

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

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

Endutent closes the currently open file.

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

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

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

COMMENTS

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

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

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

#include <search.h>

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

int hcreate (nel)
unsigned nel;

void hdestroy ( )

DESCRIPTION

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

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

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

NOTES

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

DIV — Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR — Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* (see *string*(3C)).

CHAINED — Use a linked list to resolve collisions. If this option is selected, the following other options become available.

START — Place new entries at the beginning of the linked list (default is at the end).

SORTUP — Keep the linked list sorted by key in ascending order.

SORTDOWN — Keep the linked list sorted by key in descending order.

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

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash

table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it out.

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

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

main( )
{
        /* space to store strings */
        char string_space[NUM_EMPL*20];
        /* space to store employee info */
        struct info info_space[NUM_EMPL];
        /* next avail space in string_space */
        char *str_ptr = string_space;
        /* next avail space in info_space */
        struct info *info_ptr = info_space;
        ENTRY item, *found_item, *hsearch( );
        /* name to look for in table */
        char name_to_find[30];
        int i = 0;

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

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

SEE ALSO

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

DIAGNOSTICS

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

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

WARNING

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

BUGS

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

NAME

hypot – Euclidean distance function

SYNOPSIS

#include <math.h>

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

DESCRIPTION

Hypot returns

sqrt(x * x + y * y),

taking precautions against unwarranted overflows.

DIAGNOSTICS

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

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

SEE ALSO

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

NAME

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

SYNOPSIS

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

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

DESCRIPTION

L3tol converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

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

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

SEE ALSO

fs(4).

BUGS

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

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If TYPE(*ldptr*) is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead*.

Ldahread returns SUCCESS or FAILURE. *Ldahread* will fail if TYPE(*ldptr*) does not represent an archive file, or if it cannot read the archive header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4), ar(4).

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldopen(3X) and *ldclose* are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If TYPE(*ldptr*) does not represent an archive file, *ldclose* will close the file and free the memory allocated to the LDFILE structure associated with *ldptr*. If TYPE(*ldptr*) is the magic number of an archive file, and if there are any more files in the archive, *ldclose* will reinitialize OFFSET(*ldptr*) to the file address of the next archive member and return FAILURE. The LDFILE structure is prepared for a subsequent *ldopen*(3X). In all other cases, *ldclose* returns SUCCESS.

Ldaclose closes the file and frees the memory allocated to the LDFILE structure associated with *ldptr* regardless of the value of TYPE(*ldptr*). *Ldaclose* always returns SUCCESS. The function is often used in conjunction with *ldaopen*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* into the area of memory beginning at *filehead*.

Ldfhread returns **SUCCESS** or **FAILURE**. *Ldfhread* will fail if it cannot read the file header.

In most cases the use of *ldfhread* can be avoided by using the macro **HEADER(***ldptr***)** defined in **ldfcn.h** (see*ldfcn*(4)). The information in any field, *fieldname*, of the file header may be accessed using **HEADER(***ldptr***).***fieldname*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetname – retrieve symbol name for common object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

DESCRIPTION

Ldgetname returns a pointer to the name associated with **symbol** as a string. The string is contained in a static buffer local to *ldgetname* that is overwritten by each call to *ldgetname*, and therefore must be copied by the caller if the name is to be saved.

As of UNIX system release 5.0, which corresponds to the first release of the operating system, the common object file format has been extended to handle arbitrary length symbol names with the addition of a "string table". *Ldgetname* will return the symbol name associated with a symbol table entry for either a pre-UNIX system 5.0 object file or a UNIX system 5.0 object file. Thus, *ldgetname* can be used to retrieve names from object files without any backward compatibility problems. *Ldgetname* will return NULL (defined in **stdio.h**) for a UNIX system 5.0 object file if the name cannot be retrieved. This situation can occur:

- if the "string table" cannot be found,
- if not enough memory can be allocated for the string table,
- if the string table appears not to be a string table (for example, if an auxiliary entry is handed to *ldgetname* that looks like a reference to a name in a non-existent string table), or
- if the name's offset into the string table is past the end of the string table.

Typically, *ldgetname* will be called immediately after a successful call to *ldtbread* to retrieve the name associated with the symbol table entry filled by *ldtbread*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>

int ldlread(ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO linent;

int ldlinit(ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem(ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO linent;
```

DESCRIPTION

Ldlread searches the line number entries of the common object file currently associated with *ldptr*. *Ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, the index of its entry in the object file symbol table. *Ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlinit and *ldlitem* together perform exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit*, *ldlitem* may be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply locates the line number entries for the function identified by *fcnindx*. *Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlread, *ldlinit*, and *ldlitem* each return either SUCCESS or FAILURE. *Ldlread* will fail if there are no line number entries in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*. *Ldlinit* will fail if there are no line number entries in the object file or if *fcnindx* does not index a function entry in the symbol table. *Ldlitem* will fail if it finds no line number equal to or greater than *linenum*.

The programs must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek,ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnlseek seeks to the line number entries of the section specified by *sectname*.

Ldlseek and *ldnlseek* return SUCCESS or FAILURE. *Ldlseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnlseek* will fail if there is no section name corresponding with **sectname*. Either function will fail if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

Ldohseek returns **SUCCESS** or **FAILURE**. *Ldohseek* will fail if the object file has no optional header or if it cannot seek to the optional header.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen – open a common object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

DESCRIPTION

Ldopen and *ldclose*(3X) are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus an archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value NULL, then *ldopen* will open *filename* and allocate and initialize the LDFILE structure, and return a pointer to the structure to the calling program.

If *ldptr* is valid and if TYPE(*ldptr*) is the archive magic number, *ldopen* will reinitialize the LDFILE structure for the next archive member of *filename*.

Ldopen and *ldclose*(3X) are designed to work in concert. *Ldclose* will return FAILURE only when TYPE(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, in particular whenever a new *filename* is opened, *ldopen* should be called with a NULL *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose*(3X).

```
/* for each filename to be processed */
ldptr = NULL;
do
{
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )
        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not NULL, *ldaopen* will open *filename* anew and allocate and initialize a new LDFILE structure, copying the TYPE, OFFSET, and HEADER fields from *oldptr*. *Ldaopen* returns a pointer to the new LDFILE structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both *ldopen* and *ldaopen* open *filename* for reading. Both functions return NULL if *filename* cannot be opened, or if memory for the LDFILE structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnrseek seeks to the relocation entries of the section specified by *sectname*.

Ldrseek and *ldnrseek* return SUCCESS or FAILURE. *Ldrseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnrseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

Ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

Ldshread and *ldnshread* return SUCCESS or FAILURE. *Ldshread* will fail if *sectindx* is greater than the number of sections in the object file; *ldnshread* will fail if there is no section name corresponding with *sectname*. Either function will fail if it cannot read the specified section header.

Note that the first section header has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnsseek seeks to the section specified by *sectname*.

Ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. *Ldsseek* will fail if *sectindx* is greater than the number of sections in the object file; *ldnsseek* will fail if there is no section name corresponding with *sectname*. Either function will fail if there is no section data for the specified section or if it cannot seek to the specified section.

Note that the first section has an index of *one*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* may be used in subsequent calls to *ldtbread*(3X). However, since *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it will return the index of the next entry.

Ldtbindex will fail if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

DESCRIPTION

Ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

Ldtbread returns SUCCESS or FAILURE. *Ldtbread* will fail if *symindex* is greater than the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note that the first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldfcn(4).

NAME

ldtbseek – seek to the symbol table of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek returns **SUCCESS** or **FAILURE**. *Ldtbseek* will fail if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

Logname returns a pointer to the null-terminated login name; it extracts the $LOGNAME variable from the user's environment.

This routine is kept in **/lib/libPW.a**.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

BUGS

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

This method of determining a login name is subject to forgery.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

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

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

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

DESCRIPTION

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

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

NOTES

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

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

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

EXAMPLE

This fragment will read in $\leq$ TABSIZE strings of length $\leq$ ELSIZE and store them in a table, eliminating duplicates.

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

#define TABSIZE 50
#define ELSIZE 120

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

SEE ALSO

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

DIAGNOSTICS

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

BUGS

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

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

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

void free (ptr)
char *ptr;

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

SEE ALSO

brk(2), malloc(3X).

DIAGNOSTICS

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

NOTE

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

NAME

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 (max)
int max;
```

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "–lmalloc" is used with *cc*(1) or *ld*(1).

Malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, and its contents have been destroyed (but see *mallopt* below for a way to change this behavior).

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

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

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

Mallopt provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST — Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 0.

M_NLBLKS — Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *Numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN — Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *Grain* must be greater than 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

M_KEEP — Preserve data in a freed block until the next *malloc*, *realloc*, or *calloc*. This option is provided only for compatibility with the old version of *malloc* and is not recommended.

These values are defined in the <*malloc.h*> header file.

Mallopt may be called repeatedly, but may not be called after the first small block is allocated.

Mallinfo provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo  {
        int arena;      /* total space in arena */
        int ordblks;    /* number of ordinary blocks */
        int smblks;     /* number of small blocks */
        int hblkhd;     /* space in holding block headers */
        int hblks;      /* number of holding blocks */
        int usmblks;    /* space in small blocks in use */
        int fsmblks;    /* space in free small blocks */
        int uordblks;   /* space in ordinary blocks in use */
        int fordblks;   /* space in free ordinary blocks */
        int keepcost;   /* space penalty if keep option */
                        /* is used */
}
```

This structure is defined in the <*malloc.h*> header file.

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

SEE ALSO

brk(2), malloc(3C).

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

WARNINGS

This package usually uses more data space than *malloc*(3C).
The code size is also bigger than *malloc*(3C).
Note that unlike *malloc*(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.
Undocumented features of *malloc*(3C) have not been duplicated.

NAME

matherr – error-handling function

SYNOPSIS

#include <math.h>

int matherr (x)
struct exception *x;

DESCRIPTION

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

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

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

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

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

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

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

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case DOMAIN:
                /* change sqrt to return sqrt(-arg1), not 0 */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
                }
        case SING:
                /* all other domain or sing errors, print message and abort */
```

```
            fprintf(stderr, "domain error in %s\n", x->name);
            abort( );
        case PLOSS:
            /* print detailed error message */
            fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                    x->name, x->arg1, x->retval);
            return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
}
```

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	–	–	–	–	M, 0	*
y0, y1, yn (arg $\leq$ 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.

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 efficiently 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* .

NOTE

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

BUGS

Memcmp uses native character comparison, which is signed on some machines but not on others. ASCII values are always positive, so programs that compare only ASCII values are portable.

Overlapping moves may yield surprises.

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

DESCRIPTION

Mktemp replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing **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.

SEE ALSO

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

BUGS

It is possible to run out of letters.

NAME

monitor – prepare execution profile

SYNOPSIS

#include <mon.h>

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

DESCRIPTION

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

Monitor is an interface to *profil*(2). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* WORDs (defined in the <*mon.h*> header file). *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *Lowpc* may not equal 0 for this use of *monitor*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option **–p** of *cc*(1) are recorded. (The C Library and Math Library supplied when **cc –p** is used also have call counts recorded.)

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

To profile the entire program, it is sufficient to use

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

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

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

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

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

FILES

mon.out
/lib/libp/libc.a
/lib/libp/libm.a

SEE ALSO

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

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl;

DESCRIPTION

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

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

NOTES

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

SEE ALSO

a.out(4).

DIAGNOSTICS

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

Nlist returns –1 upon error; otherwise it returns 0.

NAME

ocurse – optimized screen functions

SYNOPSIS

#include <ocurse.h>

DESCRIPTION

Ocurse is the old Berkeley curses library that uses *termcap (4)*.

These functions optimally update the screen.

Each *curses* program begins by calling *initscr* and ends by calling *endwin*.

Before a program can change a screen, it must specify the changes. It stores changes in a variable of type **WINDOW** by calling *curses* functions with the variable as argument. Once the variable contains all the changes desired, the program calls *wrefresh* to write the changes to the screen.

Most programs only need a single **WINDOW** variable. *Curses* provides a standard **WINDOW** variable for this case and a group of functions that operate on it. The variable is called *stdscr*; its special functions have the same name as the general functions minus the initial w.

FILES

/usr/include/ocurse.h – header file
/usr/lib/libocurse – curses library
/usr/lib/libtermcap – termcap library, used by curses

SEE ALSO

Ken Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package.* Available from Computer Center Library, University of California at Berkeley.
stty(2), setenv(3), termcap(4)

FUNCTIONS

addch(ch)	Add a character to *stdscr*.
addstr(str)	Add a string to *stdscr*.
box(win,vert,hor)	Draw a box around a window.
crmode()	Set cbreak mode.
clear()	Clear *stdscr*.
clearok(scr,boolf)	Set clear flag for *scr*.
clrtobot()	Clear to bottom on *stdscr*.
clrtoeol()	Clear to end of line on *stdscr*.
delch()	Delete a character.
deleteln()	Delete a line.
delwin(win)	Delete *win*.
echo()	Set echo mode.
endwin()	End window modes.
erase()	Erase *stdscr*.
getch()	Get a char through *stdscr*.
getcap(name)	Get terminal capability *name*.
getstr(str)	Get a string through *stdscr*.
gettmode()	Get tty modes.
getyx(win,y,x)	Get (y,x) co-ordinates.
inch()	Get char at current (y,x) co-ordinates.
initscr()	Initialize screens.
insch(c)	Insert a char.
insertln()	Insert a line.
leaveok(win,boolf)	Set leave flag for *win*.
longname(termbuf,name)	Get long name from *termbuf*.
move(y,x)	Move to (y,x) on *stdscr*.

mvcur(lasty,lastx,newy,newx)	Actually move cursor.
newwin(lines,cols,begin_y,begin_x)	Create a new window.
nl()	Set newline mapping.
nocrmode()	Unset cbreak mode.
noecho()	Unset echo mode.
nonl()	Unset newline mapping.
noraw()	Unset raw mode.
overlay(win1,win2)	Overlay win1 on win2.
overwrite(win1,win2)	Overwrite win1 on top of win2.
printw(fmt,arg1,arg2,...)	Printf on *stdscr*.
raw()	Set raw mode.
refresh()	Make current screen look like *stdscr*.
resetty()	Reset tty flags to stored value.
savetty()	Stored current tty flags.
scanw(fmt,arg1,arg2,...)	Scanf through *stdscr*.
scroll(win)	Scroll *win* one line.
scrollok(win,boolf)	Set scroll flag.
setterm(name)	Set term variables for name.
standend()	End standout mode.
standout()	Start standout mode.
subwin(win,lines,cols,begin_y,begin_x)	Create a subwindow.
touchwin(win)	change all of *win*.
unctrl(ch)	Printable version of *ch*.
waddch(win,ch)	Add char to *win*.
waddstr(win,str)	Add string to *win*.
wclear(win)	Clear *win*.
wclrtobot(win)	Clear to bottom of *win*.
wclrtoeol(win)	Clear to end of line on *win*.
wdelch(win,c)	Delete char from *win*.
wdeleteln(win)	Delete line from *win*.
werase(win)	Erase *win*.
wgetch(win)	Get a char through *win*.
wgetstr(win,str)	Get a string through *win*.
winch(win)	Get char at current (y,x) in *win*.
winsch(win,c)	Insert char into *win*.
winsertln(win)	Insert line into *win*.
wmove(win,y,x)	Set current (y,x) co-ordinates on *win*.
wprintw(win,fmt,arg1,arg2,...)	Printf on *win*.
wrefresh(win)	Make screen look like *win*.
wscanw(win,fmt,arg1,arg2,...)	Scanf through *win*.
wstandend(win)	End standout mode on *win*.
wstandout(win)	Start standout mode on *win*.

NAME

ofCreate, ofChangeFileLength, ofDelete — Allocate RTOS files

SYNOPSIS

```
ofCreate(pbFileSpec, cbFileSpec, pbPassword, cbPassword, lfaFileSize)
char *pbFileSpec;
shortcbFileSpec;
char *pbPassword;
shortcbPassword;
long lfaFileSize;

ofChangeFileLength(fh, lfaNewFileSize)
shortfh;
long lfaNewFileSize;

ofDelete(fh)
shortfh;
```

DESCRIPTION

OfCreate calls the RTOS CreateFile service, which creates a RTOS file. Arguments are:

- *PbFileSpec* and *cbFileSpec* specify the location and size of the new file's name. Operating system processes lack a RTOS default path, so the name must begin with a volume name in square brackets, [...], and a directory name in angle brackets, <...>. The specified volume and directory must already exist. The file name that follows the volume and directory specifications can be up to 50 characters: uppercase and lowercase letters, digits, periods (.), hyphens (-), and right angle brackets (>). Here is an example with everything:

  [sys]<sys>Big1.subd>doc-Old

  OfCreate fails if the specified directory already has a file with the specified name. RTOS does not consider two file names distinct if they differ only in the case of their letters. However, a RTOS directory preserves the case of letters as specified by *ofCreate*.

- *PbPassword* and *cbPassword* specify the location and size of the password that authorizes creation of the file. This password must match the volume or directory password. If volume or directory lacks a password, no password is needed; set *cbPassword* to 0 and *PbPassword* to anything. (To give the file itself a password, see see *ofstatus*(3X).)

- *LfaFileSize* is the initial size of the file. The size must be a multiple of 512.

See *ofopenfile*(3X) to provide a file handle for a newly-created file.

OfChangeFileLength calls the RTOS **ChangeFileLength** service, which resets the length of a file. Arguments are:

- *Lh* is a file handle returned by *ofOpen*.

- *LfaNewFileSize* is the new size of the file. The size must be a multiple of 512.

OfDelete calls the RTOS **DeleteFile** service, which deletes a file. *Fh* is a file handle returned by an *ofOpen* in modify mode.

The program must be loaded with the library flag **–lctos**.

SEE ALSO

RTOS Operating System Manual, "File Management."
ofopenfile(3X), *ofread*(3X), *ofdir*(3X), *ofstatus*(3X), *ofrename*(3X).

RETURN VALUE

0 indicates success. *OfCreate* returns 224 if the file already exists. For other errors, see

Appendix A in the *RTOS Operating System Manual.*

WARNING

Frequent calls to *OfOpen* and *CloseFile* on a nearly full volume result in files whose contents are scattered about the disk. RTOS must add additional header blocks to the disk to keep track of the fragments. Frequent calls to *ofChangeFileLength* can have the same effect.

NAME

ofCrDir, ofDIDir, ofReadDirSector — RTOS directory functions

SYNOPSIS

```
ofCrDir(pbDirSpec, cbDirSpec, pbVolPassword, cbVolPassword,
      pbDirPassword, cbDirPassword, cSectors,
      defaultFileProtectionLevel)
char  *pbDirSpec;
short cbDirSpec;
char  *pbVolPassword;
short cbVolPassword;
char  *pbDirPassword;
short cbDirPassword;
short cSectors;
short defaultFileProtectionLevel;

ofDlDir(pbDirSpec, cbDirSpec, pbPassword, cbPassword)
char  *pbDirSpec;
short cbDirSpec;
char  *pbPassword;
short cbPassword;

ofReadDirSector(pbDirSpec, cbDirSpec, pbPassword, cbPassword,
      iSector, pBufferRet)
char  *pbDirSpec;
short cbDirSpec;
char  *pbPassword;
short cbPassword;
short iSector;
char  *pBufferRet;
```

DESCRIPTION

OfCrDir calls the RTOS **CreateDir** service, which creates a RTOS directory. It takes the following arguments:

- *PbDirSpec* and *cbDirSpec* specify the location and size of the directory name. Operating system processes lack a RTOS default path, so the name must begin with a volume name in square brackets ([...]). Angle brackets around the directory name (<...>) are optional. The specified volume must already exist. The directory name that follows the volume specification can be up to 12 characters: uppercase and lowercase letters, digits, periods (.), and hyphens (-). Here is an example with everything:

  [sys]<PM.M-Changes>

  OfCrDir fails if the specified volume already has a directory with the specified name. RTOS does not consider two directory names as distinct if they differ only in the case of their letters. However, the RTOS volume control structures preserves the case of letters as specified by *ofCrDir*.

- *PbVolPassword* and *cbVolPassword* specify the location and size of a password to be compared with the volume password. If the volume lacks a password, set *cbVolPassword* to 0 and *pbVolPassword* to anything.

- *PbDirPassword* and *cbDirPassword* specify the location and size of the password to be assigned to the directory. If the directory is to have no password, set *cbDirPassword* to 0 and *pbDirPassword* to anything.

- *CSectors* is the size of the directory in sectors. In general, one sector can store information on 15 files, but this depends on the length of the file names.
- *DefaultFileProtectionLevel* indicates the initial protection of files in the directory.

OfDlDir calls the RTOS **DeleteDir** service, which deletes an empty directory. Delete or move all files from a directory before deleting the directory. *OfDlDir* takes the following arguments:

- *PbDirSpec* and *cbDirSpec* specify the location and size of the directory name. This name follows the same conventions used by *ofCrDir*.
- *PbPassword* and *cbPassword* specify the location and size of the password that authorizes the deletion of the directory. This password must match the volume password or the directory password. If volume or directory lack a password, no password is required to delete the directory: set *cbPassword* to 0 and *pbPassword* to anything.

OfReadDirSector calls the RTOS **ReadDirSector** service, which reads a single 512-byte directory sector. It takes the following arguments.

- *PbDirSpec* and *cbDirSpec* specify the location and size of the directory name. This name follows the same conventions used by *ofCrDir*.
- *PbPassword* and *cbPassword* specify the location and size of the password that authorizes access of the directory. This password must match the volume password or the directory password. If volume or directory lack a password, no password is required to delete the directory: set *cbPassword* to 0 and *pbPassword* to anything.
- *ISector* specifies which sector to read. Sectors are numbered from 0.
- *PBufferRet* points to a 512-byte area that will receive the sector.

The program must be loaded with the library flag **–lctos**.

SEE ALSO

RTOS Operating System Manual, "File Management."
ofcreate(3X) *ofopenfile*(3X) *ofread*(3X) *ofstatus*(3X) *ofrename*(3X)

RETURN VALUE

0 indicates success. *OfCrDir* returns 240 ("Directory already exists") if the specified volume alread has a directory with the specified name. *OfDlDir* returns 241 ("Directory not empty") if the directory still has files in it. For other errors, see Appendix A in the *RTOS Operating System Manual.*

NAME

ofOpenFile, ofCloseFile, ofCloseAllFiles – Access RTOS files

SYNOPSIS

```
ofOpenFile(pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode)
short*pFhRet;
char *pbFileSpec;
shortcbFileSpec;
char *pbPassword;
shortcbPassword;
shortmode;

ofCloseFile(fh)
shortfh;

ofCloseAllFiles()
```

DESCRIPTION

OfOpenFile calls the RTOS **OpenFile** service, which opens an existing RTOS file. *OfOpenFile* takes the following arguments.

- *PFhRet* specifies where *ofOpenFile* is to return the file handle. This value is similar in use to a CTIX file descriptor. Functions that do I/O, reallocate, and delete files require a valid file handle.
- *PbFileSpec* and *cbFileSpec* specify the location and length of the file name. Operating system processes lack a RTOS default path, so the name must begin with a volume name in square brackets, [...], and a directory name in angle brackets, <...>. The remainder of the name must match a name in the specfied directory, except that letters in the two names can differ in case. (See *ofcreate*(3C).)
- *PbPassword* and *cbPassword* specify the location and size of a password that authorizes access to the file. The password required depends on the protection level of the file; see Table 14-1 in the *RTOS Operating System Manual.* (Level 15 requires no password.) If no password is needed, set *cbPassword* to 0 and *PbPassword* to anything.
- *Mode* specifies the access mode: 'mr' for reading, 'mm' for modifying.

A process that has file open in modify mode is the only process that can have the file open at all. An attempt to open a file in *modify* mode will fail if *any* other process already has that file open. An attempt to open a file in *any* mode will fail if another process already has that file open in *modify* mode.

Suppose we want to open for reading a file on volume "sys" and directory "sys" called "jonah.user". The following example works if no password is required.

```
fnmp = "[sys]<sys>jonah.user";
if ((erc = ofOpenFile(&jhandle, fnmp, strlen(fnmp), 0, 0, 'mr'))
   != 0)
printf("CTOS open error %d\n", erc);
```

OfCloseFile calls the RTOS service **CloseFile** which closes a file. *Fh* is a file handle previously provided by *ofOpenFile*.

ofCloseAllFiles closes all the process's RTOS files.

FILES

SEE ALSO

RTOS Operating System Manual, "File Management."
ofcreate(3C) *ofread*(3C) *ofdir*(3C) *ofstatus*(3C) *ofrename*(3C) *ofdir*(3C)

RETURN CODE

0 indicates success. If a modify mode *ofOpenFile* returns 220 ("File in use"), some other process has the file open for reading or modifying. If a read mode *ofOpenFile* returns 220, some other process has the file open for modifying. For other errors, see Appendix A in the *RTOS Operating System Manual.*

NAME

ofRead, ofWrite – Input/output on a RTOS file

SYNOPSIS

```
ofRead(fh, pBufferRet, sBufferMax, lfa, psDataRet)
shortfh;
char *pBufferRet;
shortsBufferMax;
long lfa;
char *psDataRet;

ofWrite(fh, pBuffer, sBuffer, lfa, psDataRet)
shortfh;
char *pBuffer;
shortsBuffer;
long lfa;
char *psDataRet;
```

DESCRIPTION

OfRead calls the RTOS service **Read** which inputs one or more sectors from a RTOS file. It takes the following arguments:

- *Fh* is a file handle previously returned by *ofOpen*(3X).
- *PBufferRet* points to a region large enough to hold the sector(s) read. The region must be on an even address; a union with a **short int** will force this.
- *SBufferMax* is the number of bytes desired. This must be a multiple of 512.
- *Lfa* is the offset, from the beginning of the file, of the first byte to be read. This must be a multiple of 512.
- *PsDataRet* Indicates where *ofRead* is to return the number of bytes actually read.

ofWrite calls the RTOS service **Write ,** which ouputs one or more sectors. It takes the following arguments:

- *Fh* is a file handle previously returned by *OpenFile*.
- *PBuffer* points to the data to be output. The data must begin at an even address.
- *SBuffer* indicates the number of bytes to be output. This must be a multiple of 512.
- indicates the offset, from the beginning of the file, to which the data is to be written. This must be a multiple of 512.
- *PsDataRet* indicates where *ofWrite* is to return the number of bytes actually written.

The program must be loaded with the library flag **–lctos**.

SEE ALSO

RTOS Operating System Manual, "File Management."

ofcreate(3X) *ofopen*(3X) *ofdir*(3X) *ofstatus*(3X) *ofrename*(3X)

RETURN CODE

0 indicates success. *OfWrite* returns 2 ("End of medium") if you attempt to write past the end of the file. For other errors, see Appendix A in the *RTOS Operating System Manual.*

WARNING

If a RTOS process has written (or will read) binary integers to (from) the file, it stored (expects) them with Intel byte-ordering. See *swapshort*(3).

NAME

ofRename – rename a RTOS file

SYNOPSIS

```
ofRename(fh, pbNewFileSpec, cbNewFileSpec, pbPassword, cbPassword)
shortfh;
char *pbNewFileSpec;
shortcbNewFileSpec;
char *pbPassword;
shortcbPassword;
```

DESCRIPTION

OfRename calls the RTOS service **RenameFile**, which renames a RTOS file. It takes the following arguments:

- *Fh* is a file handle returned by a *OpenFile* in modify mode. This indicates the file to be renamed.
- *PbNewFileSpec* and *cbNewFileSpec* specify the location and size of the file's new name. The file name must include the volume and directory names. The filename conventions are the same as those for *CreateFile*(1).
- *PbPassword* and *cbPassword* specify the location and size of a password that authorizes the insertion of a file in the specified directory. This password must match the volume or directory password. If volume or directory lacks a password, no password is needed; set *cbPassword* to 0 and *PbPassword* to anything.

The program must be loaded with the library flag **–lctos**.

SEE ALSO

RTOS Operating System Manual, "File Management."

ofcreate(3X) *ofopenfile*(3X) *ofread*(3X) *ofdir*(3X) *ofstatus*(3X)

DIAGNOSTICS

0 indicates success. For errors, see Appendix A in the *RTOS Operating System Manual.*

WARNING

A rename to a new directory is meaningful; a rename to a new volume is not.

NAME

ofGetFileStatus, ofSetFileStatus – RTOS File Status

SYNOPSIS

```
ofGetFileStatus(fh, statusCode, pStatus, sStatus)
shortfh;
shortstatusCode;
char *pStatus;
shortsStatus;

ofSetFileStatus(fh, statusCode, pStatus, sStatus)
shortfh;
shortstatusCode;
char *pStatus;
shortsStatus;
```

DESCRIPTION

OfGetFileStatus and *ofSetFileStatus* call the RTOS **GetFileStatus** and **SetFileStatus** services, which get and set file information. They take the following arguments:

- *Fh* is a file handle returned by an *OpenFile* in modify mode. *StatusCode* specifies the information to be obtained or changed. *StatusCode* must be one of the following codes. *OfSetFileStatus* only sets the items marked as settable.

Code	*Item*	*Size*	*Setable?*
0	File length	4	No
1	File type	1	Yes
2	File protection level	1	Yes
3	Password	13	Yes
4	Date/time of creation	4	Yes
5	Date/time last modified	4	Yes
6	End-of-file pointer	4	Yes
7	File Header Block	512	No
8	Volume Home Block	256	No
9	Device Control Block	100	No
10	FHB Application Field	64	Yes

- *Pstatus* and *sStatus* specify the location and size of the area that holds, or is to receive, the data. If the area isn't big enough, *ofGetFileStatus* right truncates the data to fit. When setting the password, use *sStatus* to indicate the password length. When getting the password, get the password length from the first byte in the data area.

A RTOS time is represented by the following formula:

$$(d * 0x20000) + (m * 0x10000) + s$$

where d is the number of days since the beginning of March, 1952 (in the local time zone); m is 0 for midnight/AM, 1 for noon/PM; s is the number of seconds since the last midnight or noon.

The program must be loaded with the library flag **–lctos**.

SEE ALSO

RTOS Operating System Manual, "File Management."

ofcreate(3X) *ofopenfile*(3X) *ofread*(3X) *ofdir*(3X) *ofrename*(3X) *ofdir*(3X)

RETURN VALUE

0 indicates success. For errors, see Appendix A in the *RTOS Operating System Manual.*

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.

SEE ALSO

intro(2).

NAME

plot – graphics interface subroutines

SYNOPSIS

```
openpl ( )

erase ( )

label (s)
char *s;

line (x1, y1, x2, y2)
int x1, y1, x2, y2;

circle (x, y, r)
int x, y, r;

arc (x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

move (x, y)
int x, y;

cont (x, y)
int x, y;

point (x, y)
int x, y;

linemod (s)
char *s;

space (x0, y0, x1, y1)
int x0, y0, x1, y1;

closepl ( )
```

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. *Space* must be used before any of these functions to declare the amount of space necessary. See *plot*(4). *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

Circle draws a circle of radius *r* with center at the point *(x, y)*.

Arc draws an arc of a circle with center at the point *(x, y)* between the points *(x0, y0)* and *(x1, y1)*.

String arguments to *label* and *linemod* are terminated by nulls and do not contain new-lines.

See *plot*(4) for a description of the effect of the remaining functions.

The library files listed below provide several flavors of these routines.

FILES

/usr/lib/libplot.a	produces output for *tplot*(1G) filters
/usr/lib/lib300.a	for DASI 300
/usr/lib/lib300s.a	for DASI 300s
/usr/lib/lib450.a	for DASI 450
/usr/lib/lib4014.a	for TEKTRONIX 4014

WARNINGS

In order to compile a program containing these functions in *file.c* it is necessary to use "cc *file.c* -lplot".

In order to execute it, it is necessary to use "a.out | tplot".

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

SEE ALSO

graph(1G), stat(1G), tplot(1G), plot(4).

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 shell command line and an I/O mode, either **r** for reading or **w** for writing. *Popen* creates a pipe between the calling program 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).

DIAGNOSTICS

Popen returns a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

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

BUGS

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

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

#include <stdio.h>

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

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

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

DESCRIPTION

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

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

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

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

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

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 (which 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. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

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

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

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

c The character *arg* is printed.

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

% Print a %; no argument is converted.

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

EXAMPLES

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

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

To print π to 5 decimal places:

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

SEE ALSO

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

NAME

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

SYNOPSIS

```
#include <stdio.h>

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

int putchar (c)
int c;

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

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

DESCRIPTION

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

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

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

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 newline character is written or terminal input is requested). *Setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

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

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant EOF. This will occu 'f 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.

BUGS

Because it is implemented as a macro, *putc* treats incorrectly a *stream* argument with side effects. In particular, **putc(c, *f++);** doesn't work sensibly. *Fputc* should be used instead.
Because of possible differences in word length and byte ordering, files written using *putw* are machine-dependent, and may not be read using *getw* on a different processor.

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
char *string;

DESCRIPTION

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

DIAGNOSTICS

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

SEE ALSO

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

WARNINGS

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

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

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

DESCRIPTION

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

DIAGNOSTICS

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

SEE ALSO

getpwent(3C).

WARNING

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

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

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

DESCRIPTION

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

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

Neither function writes the terminating null character.

DIAGNOSTICS

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

SEE ALSO

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

NOTES

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

NAME

qsort – quicker sort

SYNOPSIS

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

DESCRIPTION

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

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

NOTES

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

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

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

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

NAME

quAdd – add a new entry to a RTOS queue

SYNOPSIS

```
quAdd(pbQueueName, cbQueueName, fQueueIfNoServer, priority,
                queueType, pEntry, sEntry, pDateTime, repeatTime)
char            *pbQueueName;
short           cbQueueName;
char            fQueueIfNoServer;
char            priority;
short           queueType;
char            *pEntry;
short           sEntry;
unsigned long*pDateTime;
short           repeatTime;
```

DESCRIPTION

QuAdd calls the RTOS AddQueueEntry service. An operating system process that wants to submit a request to a RTOS queue server creates a queue entry with *quAdd*. *QuAdd* takes the following arguments.

- *PbQueueName* and *cbQueueName* describe the location and length of of a queue name. This must be one of the queues mentioned in the RTOS file **[sys]<sys>queue.index**.
- *FQueueIfNoServer* determines the action if the queue manager finds that no servers are active for the specified queue. **0xFF** means to queue the entry anyway. **0** means abort the queue entry.
- *Priority* sets the queue entry's priority. **0** is the highest priority, **9** is the lowest.
- *QueueType* is the type of queue. This must match the number given in the fourth field of the queue's entry is the queue index file.
- *PEntry* and *sEntry* describe the size and location of entry data. The size and layout of this data area is conventional for each queue.
- *PDateTime* points to the service time. A server will serve the request no sooner than the service time.

  The service time must be in RTOS format:

  $$(d * 0x20000) + (m * 0x10000) + s$$

  where *d* is the number of days since the beginning of March, 1952 (in the local time zone); *m* is 0 for midnight/AM, 1 for noon/PM; *s* is the number of seconds since the last midnight or noon.

  A service time of 0 means "undated"; the queue manager provides servers for all undated requests before it provides servers for any dated requests.
- *RepeatTime* specifies a repeat interval. Unless this value is 0, the queue manager resubmits the request *RepeatTime* minutes after a queue server deletes it. Thus the request repeats forever, with at least *RepeatTime* minutes between repetitions. An operating system process can terminate this loop with *quRemove*(3X).

Queue servers run under RTOS and thus expect integers to have Intel byte-ordering. *QuAdd* translates *queueType*, the date, and *repeatTime*, but does nothing about entry data. To translate entry data, see *swapshort*(3).

The program must be loaded with the library flag **–lctos**.

FILES

[sys]<sys>queue.index - master queue index

SEE ALSO

quremove(3X), *quread*(3X).

RTOS Operating System Manual, "Queue Management."

RETURN VALUE

0 indicates success. 254 ("Queue not served") if *fQueueIfNoServer* is 0 and no servers are active on the specified queue.

NAME

quReadNext, quReadKeyed – examine RTOS queue

SYNOPSIS

```
structQueueStatusBlock {
      long qehRet;
      char priority;
      char padding;
      shortServerUserNumber;
      long qehNextRet;
      };

quReadNext(pbQueueName, cbQueueName, qeh,
      pEntryRet, sEntryRet, pStatusBlock, sStatusBlock)
char  *pbQueueName;
short cbQueueName;
long  qeh;
char  *pEntryRet;
short sEntryRet;
structQueueStatusBlock *pStatusBlock;
short sStatusBlock;

quReadKeyed(pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1,
      pbKey2, cbKey2, oKey2, pEntryRet, sEntryRet,
      pStatusBlock, sStatusBlock)
char  *pbQueueName;
short cbQueueName;
char  *pbKey1;
short cbKey1;
short oKey1;
char  *pbKey2;
short cbKey2;
short oKey2;
char  *pEntryRet;
short sEntryRet;
structQueueStatusBlock *pStatusBlock;
short sStatusBlock;
```

DESCRIPTION

QuReadNext and *QuReadKeyed* call the RTOS services **ReadNextQueueEntry** and **ReadKeyedQueueEntry**. A queue client uses *quReadNext* or *quReadKeyed* to examine a RTOS queue. Each call returns information on a single queue entry. *QuReadNext* and *quReadKeyed* have the following arguments in common.

- *PbQueueName* and *cbQueueName* describe the location and size of a queue name.
- *PEntryRet* and *sEntryRet* describe the location and size of an area that is to receive entry data. Size and layout of entry data is specific to each queue. If the area is smaller than an entry's data, the data is right-truncated to fit.
- *PStatusBlock* and *sStatusBlock* describe the location and size of an area that is to receive the entry's status block. If the area is smaller than **sizeof(QueueStatusBlock)** the block is right-truncated to fit.

QuReadNext and *quReadKeyed* return the following values in the status block.

- *QehRet* is the queue entry handle. This integer value is unique for each entry in the queu.

- *Priority* is the priority of the entry.
- *ServerUserNum* is the RTOS user number of the queue server that has approrpriated (marked) the request and plans to service it. If no server has appropriated the request, *serverUserNum* is **–1**.
- *QehNextRet* is the queue entry handle for the next entry in the queue. If the current entry is the last entry in the queue, *QehNextRet* is **–1**.

The following argument is specific to *quReadNext*.

- *Qeh* specifies the queue entry to be read. 0 indicates the first queue entry; any other value must be a queue entry handle.

This example passes the data for each entry in **SPL** to **prentry()**.

```
qnl = strlen(qns == "SPL");
for (handle = 0; handle != -1; handle = status.QehNextRet) {
        quReadNext(qnl, qns, handle, &data,
                sizeof(data), &status, sizeof(status));
        prentry(&status);
}
```

The following arguments are specific to *quReadKeyed*.

- *PbKey1* and *cbKey1* describe the location and size of the first search key. If there is no first search key, set *cbKey1* to **0**.
- *OKey1* is the offset of the first search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the first search key. *QuReadKeyed* assumes that the first byte of this string gives the size of the remainder of the string. If there is no first search key, the function ignores *oKey1*.
- *PbKey2* and *cbKey2* describe the location and size of the second search key. If there is no second search key, set *cbKey2* to **0**.
- *OKey2* is the offset of the second search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the second search key. *QuReadKeyed* assumes that the first byte of this string gives the size of the remainder of the string. If there is no second search key, the function ignores *oKey2*.

The client that calls *quReadKeyed* must supply 1 or 2 search keys. *QuReadKeyed* returns the first entry that matches both search keys. If only one key is given, *QuReadKeyed* returns the first entry that matches that single key.

The program must be loaded with the library flag **–lctos**.

FILES

[sys]<sys>queue.index – master queue index

SEE ALSO

quremove(3X) *quadd*(3X)

RETURN VALUE

0 indicates success. *QuReadNext* returns 904 ("Entry deleted") if another client deletes a queue entry between the time you get the entry's handle and the time you try to read it.

NAME

quRemove – take back a RTOS queue request

SYNOPSIS

quRemove(pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2)
char *pbQueueName;
short cbQueueName;
char *pbKey1;
short cbKey1;
short oKey1;
char *pbKey2;
short cbKey2;
short oKey2;

DESCRIPTION

QuRemove calls the RTOS service **RemoveKeyedQueueEntry**. A queue client uses *quRemove* to delete entries from a RTOS queue. *quRemove* uses search keys to identify the requeust. It takes the following arguments.

- *PbQueueName* and *cbQueueName* describe the location and size of a queue name.
- *PbKey1* and *cbKey1* describe the location and size of the first search key. If there is no first search key, set *cbKey1* to **0**.
- *OKey1* is the offset of the first search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the first search key. *quRemove* assumes that the first byte of this string gives the size of the remainder of the string. If there is no first search key, the function ignores *oKey1*.
- *PbKey2* and *cbKey2* describe the location and size of the second search key. If there is no second search key, set *cbKey2* to **0**.
- *OKey2* is the offset of the second search string. This is the offset, from the beginning of the entry data, of a string that is to be compared with the second search key. *quRemove* assumes that the first byte of this string gives the size of the remainder of the string. If there is no second search key, the function ignores

The client that calls *quRemove* must supply 1 or 2 search keys. *quRemove* deletes the first entry that matches both search keys. If only one key is given, *quRemove* deletes the first entry that matches that single key, *oKey2*.

The program must be loaded with the library flag **–lctos**.

FILES

[sys]<sys>queue.index – master queue index

SEE ALSO

readqueue(3X) *addqueue*(3X)

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned seed;

DESCRIPTION

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

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

NOTE

The spectral properties of *rand* leave a great deal to be desired. *Drand48*(3C) provides a much better, though more elaborate, random-number generator.

SEE ALSO

drand48(3C).

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*(3C) 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. *Regcmp*(1) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer *__loc1* points to where the match began. *Regcmp* and *regex* were mostly borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[ ] * . ^ These symbols retain their current meaning.

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

– Within brackets the minus means *through*. For example, [**a–z**] is equivalent to [**abcd . . .xyz**]. The – can appear as itself only if used as the first or last character. For example, the character class expression []–] matches the characters] and –.

+ A regular expression followed by + means *one or more times*. For example, [**0–9**]+ is equivalent to [**0–9**][**0–9**]*.

{m} {m,} {m,u}
Integer values enclosed in { } indicate the number of times the preceding regular expression is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(. . .)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(. . .) Parentheses are used for grouping. An operator, e.g., *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

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 (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        . . .
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(1)) against *string*.

This routine is kept in **/lib/libPW.a**.

SEE ALSO

ed(1), regcmp(1), malloc(3C).

BUGS

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*(3C) 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;
}
```

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.

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.

e,f,g a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional +, –, or space, followed by an integer.

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

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

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

The conversion characters **d**, **u**, **o**, 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 EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

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

EXAMPLES

The call:

```
int i, 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**.

SEE ALSO

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

NOTE

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

DIAGNOSTICS

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

BUGS

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

NAME

setbuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

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

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

DESCRIPTION

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

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

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

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

_IOFBF causes input/output to be fully buffered.

_IOLBF causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

_IONBF causes input/output to be completely unbuffered.

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

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

SEE ALSO

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

DIAGNOSTICS

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

NOTE

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

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

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

DESCRIPTION

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

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

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

SEE ALSO

signal(2).

WARNING

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

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

DESCRIPTION

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

DIAGNOSTICS

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

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

SEE ALSO

matherr(3M).

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

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

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

SEE ALSO

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

NAME

spawnlp, spawnvp – execute a process on a specific Application Processor

SYNOPSIS

```
int
spawnlp(apnum, directory,  name, arg0, arg1, ..., argn, 0)
int apnum;
char *directory;
char *name, *arg0, *arg1, ..., *argn;

int
spawnvp(apnum, directory, name, argv)
int apnum;
char *directory;
char *name, *argv[];

extern char **environ;
```

DESCRIPTION

The *spawn* functions, *spawnlp* and *spawnvp*, execute a file on the specified System 6600 Application Processor, creating a new process on that Processor. The practical effect is that of a *fork/exec* sequence with the following differences:

- *Spawn* will create the new process on any Application Processor. *Fork/exec* always creates the new process on the parent process's Application Processor.
- A *spawn* process is not a child of the process that called *spawn*; it is a child of the spawn server on the designated Application Processor (*spawnsrv*(1M)). Thus the process that called *spawn* cannot *wait*(2) for the new process's death; use *spwait*(3X) instead. Also, not all the attributes that are inherited accross a *fork* are inherited across a *spawn*.
- A *fork/exec* is less expensive than a *spawn*.

The spawn server passes the following attributes to the new process, based partially on the attributes of the calling process:

- File descriptors 0, 1, and 2 (standard input, output, and error) of the new process are open to **/dev/null**. None of the calling process's file descriptors are available to the new process.
- Signals caught by the calling process terminate the new process. Other signals (ignored by or causing termination of the calling process) have the same effect on the new process they had on the calling process.
- The new process inherits the following from the calling process, unchanged: environment parameters (variables); file creation mask (*umask*(2)); effective user ID and group ID.
- If the calling process's effective user ID is 0, the new process inherits the calling process's real user ID and group ID. Otherwise, the new process's real IDs are the same as its effective IDs.

The calling conventions for *spawnlp* and *spawnvp* are the same as for *execlp* and *execvp* (*exec*(2)), but with two additional parameters at the beginning:

apnum The number of the Application Processor that is to run the new process. Application Processors are numbered from 0. Viewed from behind, Application Processors in the rightmost enclosure are counted first, working left; within an enclosure, count left-to-right. See the *System 6600 Administrator's Guide.*

directory

A pointer to a null-terminated string identifying the new process's working directory. If *directory* is **(char *) 0** , (**NULL** in **<stdio.h>**) the new process's working directory is the same as the calling process's. (Use of **NULL** is expensive: it causes a call to

cwd(3).)

RETURN VALUE

Both functions return –1 on error; otherwise they return the process number of the new process.

SEE ALSO

apnum(1), pwd(1), spawn(1), apnum(2), fork(2), signal(2), getcwd(3C), spwait(3X), environ(5).

EXAMPLES

The following runs "myprog" in the same directory as the current process, but runs it on AP 01:

```
#define NULL ((char *) 0)
spawnlp(01, NULL, "myprog", "myprog", "arg1", NULL);
```

The following runs a shell on the other AP:

```
spawnlp(01, "/", "/bin/sh", "-sh", "-c", "cd $HOME; exec myprog", NULL);
```

NAME

sputl, sgetl – access long integer data in a machine-independent fashion.

SYNOPSIS

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

DESCRIPTION

Sputl takes the four bytes of the long integer *value* and places them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

Sgetl retrieves the four bytes in memory starting at the address pointed to by *buffer* and returns the long integer value in the byte ordering of the host machine.

The combination of *sputl* and *sgetl* provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be loaded with the object-file access routine library **libld.a**.

NAME

spwait – wait for spawned process to terminate

SYNOPSIS

spwait(pid, status)
int pid, *status;

DESCRIPTION

Spwait suspends the calling process until a signal is received or the process specified by process ID *pid* terminates. The specified process must have been previously spawned (*spawn*(3X)) by the calling process.

If *status* is not equal to **(int *) 0**, the word it points to receives two data:

- The high byte gets the low byte of the specified process's *exit*(2) parameter.
- The low byte get the specified process's termination status (*signal*(2)). If the termination status's 0200 bit is set, the process produced a core image when it terminated.

SEE ALSO

spawn(1), exit(2), fork(2), signal(2), spawn(3x).

RETURN VALUE

If *spwait* returns due to the receipt of a signal, a value of –1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a terminated spawn process, the process ID of the child is returned to the calling process. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NAME

ssignal, gsignal – software signals

SYNOPSIS

```
#include <signal.h>

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

int gsignal (sig)
int sig;
```

DESCRIPTION

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

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

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

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

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

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

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

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

SEE ALSO

signal(2).

NOTES

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

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

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

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

stdin	standard input file
stdout	standard output file
stderr	standard error file

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

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

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

stdipc – standard interprocess communication package (ftok)

SYNOPSIS

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

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

DESCRIPTION

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

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

SEE ALSO

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

DIAGNOSTICS

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

WARNING

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

NAME

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, c;

char *strrchr (s, c)
char *s, 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 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 on 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.

NOTE

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

BUGS

Strcmp and *strncmp* use native character comparison, which is signed on Motorola 68000-family processors. This means that characters are 8-bit signed values; all ASCII characters have values of at least 0; non-ASCII are negative. On some machines, all characters are positive. Thus programs that only compare ASCII values are portable; programs that compare ASCII with non-ASCII values are not.

Overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

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

double atof (str)
char *str;

DESCRIPTION

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

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

long atol (str)
char *str;

int atoi (str)
char *str;

DESCRIPTION

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

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

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

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

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

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

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

SEE ALSO

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

BUGS

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

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

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP-11s and other machines. *Nbytes* should be even and non-negative. If *nbytes* is odd and positive *swab* uses *nbytes*–1 instead. If *nbytes* is negative, *swab* does nothing.

NAME

swapshort, swaplong – translate byte orders to Motorola/Intel

SYNOPSIS

swapshort(s)
short s;

swaplong(l)
long l;

DESCRIPTION

Processes that run on a System 6600 Application Processor (operating system processes) do not store integers the same way as do processes that run on other System 6600 Processors (RTOS processes). Operating system processes use Motorola ordering; RTOS processes use Intel ordering. Operating system processes must translate integers sent to or received from RTOS processes.

Library functions do this translation whenever they know an integer value is involved. For example, *AddQueueEntry* translates integers that are supplied for all queue entries: the priority, the queue type, and the data. But *AddQueueEntry* does not translate any integers in the entry data.

Swaplong translates to or from Intel four-byte integers. *Swaplong* returns *l* with its bytes in reverse order. For example, if *l* is 4885001 (0x004A8A09) *swaplong* returns 160057856 (0x098A4A00).

Swapshort translates to or from Intel two-byte integers. *Swapshort* returns *S* with its bytes in reverse order.

The program must be loaded with the **–lctos** library flag.

NAME

system – issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

DESCRIPTION

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

FILES

/bin/sh

SEE ALSO

sh(1), exec(2).

DIAGNOSTICS

System forks to create a child process that in turn exec's */bin/sh* in order to execute *string*. If the fork or exec fails, *system* returns –1 and sets *errno*.

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – terminal independent operations

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cmstr, destcol, destline)
char *cmstr;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use information from terminal descriptions that follow the conventions in *termcap*(4). The functions only do basic screen manipulation: they find and output specified terminal function strings and interpret the **cm** string. *Curses*(3X) describes a screen updating package built on *termcap*.

Tgetent finds and copies a terminal description. *Name* is the name of the description; *bp* points to a buffer to hold the description. *Tgetent* passes *bp* to the other *termcap* functions; the buffer must remain allocated until the program is done with the *termcap* functions.

Tgetent uses the **TERM** and **TERMCAP** environment variables to locate the terminal description.

- If **TERMCAP** isn't set or is empty, *tgetent* searches for *name* in */etc/termcap*.
- If **TERMCAP** contains the full pathname of a file (any string that begins with /), *tgetent* searches for *name* in that file.
- If **TERMCAP** contains any string that does not begin with / and **TERM** is not set or matches *name*, *tgetent* copies the **TERMCAP** string.
- If **TERMCAP** contains any string that does not begin with / and **TERM** does not match *name*, *tgetent* searches for *name* in */etc/termcap*.

Tgetent returns –1 if it couldn't open the terminal capability file, 0 if it couldn't find an entry for *name*, and 1 upon success.

Tgetnum returns the value of the numeric capability whose name is *id*. It returns –1 if the terminal lacks the specified capability or it is not a numeric capability.

Tgetflag returns 1 if the terminal has boolean capability whose name is *id*, 0 if it does not or it is not a boolean capability.

Tgetstr copies and interprets the value of the string capability named by *id*. *Tgetstr* expands instances in the string of \ and ^. It leaves the expanded string in the buffer *indirectly* pointed to by *area* and leaves the buffer's direct pointer pointing to the end of the expanded string; for example,

```
tgetstr("cl", &ptr);
```

where *ptr* is a character pointer -- not an array name! *Tgetstr* returns a (direct) pointer to the beginning of the string.

Tgoto interprets the % escapes in a **cm** string. It returns *cmstr* with the % sequences changed to the position indicated by *destcol* and *destline*. This function must have the external variables *BC* and *UP* set to the values of the **bc** and **up** capabilities; if the terminal lacks the capability, set the external variable to null. If *tgoto* can't interpret all the % sequences in **cm**, it returns "OOPS"

Tgoto avoids producing characters that might be misinterpreted by the terminal interface. If expanding a % sequence would produce a null, control-d, or null, the function will, if possible, send the cursor to the next line or column and use *BC* or *UP* to move to the correct location. Note that *tgoto* does not avoid producing tabs; a program must turn off the **TAB3** feature of the terminal interface (*termio*(7)). This is a good idea anyway: some terminals use the tab character as a nondestructive space.

Tputs directs the output of a string returned by *tgetstr* or *tgoto*. This function must have the external variable *PC* set to the value of the **pc** capability; if the terminal lacks the capability, set the external variable to null. *Tputs* interprets any delay at the beginning of the string. *Cp* is the string to be output; *affcnt* is the number of lines affected by the action (1 if "number of lines affected" doesn't mean anything); and *outc* points to a function that takes a single **char** argument and outputs it, such as *putchar*.

FILES

/usr/lib/libtermcap.a	library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ( )

DESCRIPTION

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

SEE ALSO

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

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>

char *tmpnam (s)
char *s;

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

DESCRIPTION

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

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

Tempnam allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string 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. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory.

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

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

NOTES

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

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

SEE ALSO

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

BUGS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.
Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or *mktemp*, and the file names are chosen so as to render duplication by other means unlikely.

NAME

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

SYNOPSIS

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double 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 π.

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 π, using the signs of both arguments to determine the quadrant of the return value.

DIAGNOSTICS

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

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

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

SEE ALSO

matherr(3M).

NAME

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

SYNOPSIS

#include <search.h>

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

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

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

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

DESCRIPTION

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

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

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

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

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

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

EXAMPLE

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

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

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

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

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, &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)
struct node *node1, *node2;
{
        return strcmp(node1->string, 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);
            }
        }
```

SEE ALSO

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

DIAGNOSTICS

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

A NULL pointer is returned by *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.

WARNINGS

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

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

BUGS

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

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
int fildes;

DESCRIPTION

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

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

FILES

/dev/*

DIAGNOSTICS

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

BUGS

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

NAME

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

SYNOPSIS

int ttyslot ( )

DESCRIPTION

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

FILES

/etc/utmp

SEE ALSO

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

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

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.

SEE ALSO

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

DIAGNOSTICS

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

NAME

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

SYNOPSIS

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

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *format;
va_list ap;
```

DESCRIPTION

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

EXAMPLE

The following demonstrates 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( );
}
```

SEE ALSO

printf(3S), varargs(5).

NAME

wmgetid – get window ID

SYNOPSIS

#include <oa/wm.h>

int wmgetid(fildes);
int fildes;

DESCRIPTION

Wmgetid returns the window ID associated with the file descriptor *fildes*. A window ID is a positive integer that identifies the window associated with the file descriptor. The ID is passed to other window management library functions to identify the particular window being acted upon. The only way to get a valid window ID is from a window management library call; do not use a value obtained any other way.

To get all the window IDs for a terminal, use the layout structure written by *wmlayout*(3X) or *wmop*(3X). To associate a file descriptor with a different window, use *wmsetid*(3X)

Wmgetid fails if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The window manager is not running on the terminal. [ENOENT]

FILES

/dev/tty*
/usr/lib/libwm.a – window management library

SEE ALSO

wm(1), wmop(3X), wmlayout(3X), wmsetid(3X).

RETURN VALUE

If success, the window ID associated with *fildes*. Otherwise, –1 is returned and *errno* is set.

NAME

wmlayout – get terminal's window layout

SYNOPSIS

```
#include <oa/wm.h>

int wmlayout(fildes, layout)
int fildes;
struct wm_layout *layout;
```

DESCRIPTION

Wmlayout fetches a description of the screen layout of a terminal under window management. *Fildes* is a file descriptor associated with the terminal's special file by an *creat*, *dup*, *fcntl*, or *open* system call; the association of *fildes* with a particular window is not used. *Layout* points to an area that is to receive the description. Before calling *wmlayout*, a program must set *layout->maxwcount* to indicate the number of window descriptions the area can accomdate; the constant WM_MAX gives the number of windows currently permitted. The description consists of the following data structures:

```
struct wm_layout {
        int     cwindowid;
        short   maxwcount;
        short   wcount;
        struct wm_wlayoutw[WM_MAX];
        };

struct wm_wlayout {
        int     windowid;
        short pwindowid;
        short startrow;
        short startcolumn;
        short drows;
        short dcolumns;
        short syncrow;
        short synccolumn;
        short vrows;
        short vcolumns;
        short crow;
        short ccolumn;
        char  reserved[6]; /* must be 0 */
        }
```

Here are the meanings of the fields in a *wm_layout* structure:

cwindowid The window ID of the active window.

maxwcount Number of window descriptions this structure has room for. Normally set to WM_MAX, so as to get all of them.

wcount Number of windows currently on terminal.

w Array of individual window descriptions.

Here are the meanings of the fields in a *wm_wlayout* structure:

windowid The window ID.

pwindowid The physical window ID. Meant only for window management internal use.

startrow Starting physical row of the window (the tag line is on the row before).

startcolumn Starting physical column of the window. Currently this value is always 1.

drows The number of displayed rows in the window. Note that the tag line is not counted in this value.

dcolumns The number of displayed columns in the window. Currently this value is always 80.

syncrow Virtual display row that corresponds to the first row of the window.

synccolumn Virtual display column that corresponds to the first column of the window. Currently this value is always 1.

vrows Number of rows in virtual display.

vcolumns Number of columns in virtual display. Currently this value is always 80.

crow The current cursor row number.

ccolumn The current cursor column number.

reserved Always zeroes.

Rows and columns are numbered from 1.

A window ID is a positive integer that identifies the window associated with the file descriptor. The ID is passed to other window management library functions to identify the particular window being acted upon. The only way to get a valid window ID is from a window management library call; do not use a value obtained any other way.

Currently, physical windows always start in column zero and physical windows and virtual displays are always 80 columns wide.

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

Fildes is not an open file descriptor. [EBADF]

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The structure pointed to by *windowreq* is invalid. [EINVAL]

The window manager is not running on the terminal. [ENOENT].

FILES

/usr/lib/libwm.a – window management library.
/dev/tty*

SEE ALSO

wm(1), wmgetid(3X), wmsetid(3), wmop(3X).

RETURN VALUE

Success returns 0; failure returns –1 and sets *errno* to indicate the error.

NAME

wmop – window management operations

SYNOPSIS

#include <oa/wm.h>

int wmop(fildes, windowreq, layout)
int fildes;
struct wm_request *windowreq;
struct wm_layout *layout;

DESCRIPTION

Wmop manipulates windows on a terminal under window management. It is normally used by application programs. *Fildes* is a file descriptor associated with the terminal's special file by an *creat*, *dup*, *fcntl*, or *open* system call; the association of *fildes* with a particular window is not used. *Windowreq* is a a pointer to a structure that describes the operation. *Layout* is an optional pointer to a layout structure of the type used by *wmlayout*; if present, the structure is filled with the new description of the window.

The request structure is defined as follows:

```
struct wm_request {
        int     request;
        int     windowid;
        int     (*notify)()
        short startrow;
        short startcolumn;
        short drows;
        short dcolumns;
        short syncrow;
        short synccolumn;
        short vrows;
        short vcolumns;
        short crow;
        short ccolumn;
};
```

Only two fields in the request structure are used by all operations:

- *Request* specifies the operations desired. *Request* is the bitwise or of the operation constants described below.
- *Windowid* specifies a window usually with a window ID returned by a previous *wmop*, *wmlayout*, or *wmgetid*. The only way to get a valid window ID is from a window management library call; do not use a value obtained any other way. If the operations do not include WM_CREATE (create a new window), *windowid* is a window ID that specifies the single window to which the operations apply. If the operations do include WM_CREATE, *windowid* must be either a window ID, indicating the window that yields space for the new window, or 0, a value with special meanings described under WM_CREATE and WM_START; the other operations apply to the new window.

WM_CREATE Create a new window. Other operations describe the new window's characteristics; if no other operations are specified with WM_CREATE, the new window has the following characteristics:

- The new window occupies the bottom half of the window specified by *windowid*. If *windowid* is 0, the new window occupies the bottom half of the active window.

- The new window's virtual display is 28 lines long.
- The cursor is on the first line of the new window's virtual display, which is also the first line of the new window.
- The user is permitted to split the new window only if the old window permitted user splits. See WM_SPLIT.

WM_DESTROY Destroy the window. If the window is the top window, the destroyed window's rows go to the window below; otherwise the destroyed window's rows go to the window above. If the destroyed window was the active window, the window that gets the destroyed window's rows is activated.

WM_DSIZE Change window size. This operation can be modified by WMDRSIZE; this description assumes it is not. The window size, which does not include the window's tag line, can vary from 0 to 26. *Drows* specifies the new window size.

If **WM_DSIZE** is specified with WM_CREATE, *drows* specifies the new window's size.

WM_DRSIZE Modifies WM_DSIZE so that *drows* specifies an offset relative the current value, rather than an absolute size. *Drows* can be negative.

If WM_DSIZE and WM_DRSIZE are specified with WM_CREATE, *drows* specifies the new window's size relative to the size of the old window. Thus in this case, *drows* must be negative.

WM_DSTART Set the starting row of the window (not the tag line, which is automatically on the row before). This operation may be modified by WM_DRSTART; this description assumes it is not. Rows are numbered from 1, and a window can start on any row from 2 to 28. *Startrow* specifies the new starting row.

IF WM_DSTART is specified with WM_CREATE and *windowid* is 0, *startrow* specifies the new window's starting position on the screen, without reference to an existing window.

WM_DRSTART Modifies WM_DSTART so that *startrow* specifies an offset relative the current value, rather than an absolute starting row. *Startrow* can be negative.

If WM_DSTART and WM_DRSTART are specified with WM_CREATE, *startrow* must be non-negative; the new window starts *startrow* rows after the start of the old window. If *startrow* is 0, the new window takes the top portion of the old window's rows instead of the bottom. If *startrow* is positive, WM_DSIZE is ineffective: the size of the new window is dictated by the size of the old.

WM_VSIZE Set virtual window size to *vrows* long. The operations can be modified by WM_VRSIZE. In any case, the virtual display must be 1 to 28 rows long.

If the virtual display is shortened past the cursor, the cursor must be moved to within the new virtual display end. If the WM_CURSOR operation is not specified at the same time, the terminal moves the cursor to the new last line of the virtual display.

WM_VRSIZE Modifies WM_VSIZE so that *vrows* is an offset to the present value. *Vrows* can be negative.

WM_VSTART Synchronize the window and its virtual display by making virtual display row *syncrow* (numbered from 1) the first row on the window. This operation can be modified by WM_VRSTART. The window manager will modify a WM_VSTART operation as necessary to keep the window from extending past the bottom of the virtual display. If the cursor is visible, the terminal

software will modify a WM_VSTART operation as necessary to keep the cursor in the window.

WM_VRSTART — Modify WM_VSTART so that *syncrow* is an offset to the present value. *Syncrow* can be negative.

WM_SELECT — Make the window the active window.

WM_DESELECT — If the window is the active window, make another window the active window: if the designated window is the top window, the window below; otherwise the window above.

WM_CURSOR — Position the cursor on row *crow*.

WM_SPLIT — Enable change of splitting permission. Used in conjunction with WM_NSPLIT. If WM_SPLIT is specified alone: the user can split the window as long as the terminal can handle another window. If WM_SPLIT and WM_NSPLIT are specified together, the SPLIT key is ineffective when the window is active.

WM_NSPLIT — Disable window split. Always used in conjunction with WM_SPLIT, which see.

WM_NOTIFY — *Notfiy* is a notify procedure. Set *notify* to (int (*)()) 0 to disable an existing notify procedure. The calling process will be interrupted and *notify* called if any other process or the user changes the status of the window. Window status includes window size, location, and whether it is active, but does not include cursor location.

Currently, all windows and displays must begin in column 0 and be 80 columns wide.

Wmop fails if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The structure pointed to by *windowreq* is invalid. [EINVAL]

The window manager is not running on the terminal. [ENOENT]

RETURN VALUE

If the operations were successful, the window ID of the affected window (the new window if one was created) is returned. Otherwise, –1 is returned and *errno* is set.

WARNINGS

Use *wmop* conservatively and with extreme care. Indescriminate use by programs competing for window space can result in race conditions and screen image instability.

The window manager and terminal software silently enforce basic consistency. A program must not make assumptions about what the window looks like after a successful *wmop*; instead it must examine the new *wmlayout* structure to find out what actually happened.

FILES

/dev/tty*
/usr/lib/libwm.a – window management library

SEE ALSO

signal(2), wmgetid(3), wmlayout(3), wmsetid(3).
ferror(3S) to get file descriptor for terminal accessed with standard input/output package.

NAME

wmsetid, wmsetids – associate a file descriptor with a window

SYNOPSIS

```
#include <oa/wm.h>

int wmsetid(fildes, windowid)
int windowid;
int fildes;

int wmsetids(fildes, windowid)
int windowid;
int fildes;
```

DESCRIPTION

Wmsetid and *wmsetids* change the window with which a file descriptor is associated. *Fildes* must be a file descriptor open to a terminal on which the window manager is running. *Fildes* becomes associated with the window (on the same terminal) indicated by *windowid*, which must be a window ID obtained from a previous *wmgetid*(3X), *wmlayout*(3X), or *wmop*(3X) call.

If a program performs a *wmsetid* on an inherited file descriptor, all processes that have inherited and use the same file descriptor and the process they inherited it from are affected. By convention, 0 (equivalent to fileno(stdin)) 1 (equivalent to fileno(stdout)) and 2 (equivalent to fileno(stderr)) are inherited file descriptors. The following code closes and reopens them so that a *wmsetid* on them doesn't affect other processes. It should be executed before terminal input/output begins.

```
tty=ttyname(0);
close(0);
close(1);
open(tty, O_RDWR);
close(2);
dup(0);
dup(0);
```

Be sure to complete buffered terminal output before switching windows. See *fclose*(3S) if you use the standard input/output package.

Wmsetid and *wmsetids* are different only when executed by a process group leader. If the process group leader calls *wmsetids* and the specified window is not already a controlling window for another process group, the specified window becomes the process group's controlling window. (For more details on control windows, see *termio*(7) and *window*(7).) *Wmsetid* never changes the controlling window under any circumstances.

Wmsetid and *wmsetids* fail if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

The indicated file does not represent a terminal, or the terminal cannot support window management. [ENOTTY]

The structure pointed to by *windowreq* is invalid. [EINVAL]

The window manager is not running on the terminal. [ENOENT]

FILES

/dev/tty*
/usr/lib/libwm.a – window management library

SEE ALSO

wm(1), wmop(3X), wmlayout(3X), wmgetid(3X).
ferror(3S) – fileno function

ttyname(3C), open(2), close(2), dup(2).

RETURN VALUES

A nonnegative value indicates success: 0 if the file descriptor wasn't associated with a window before the call, the old window ID otherwise. On error, –1 is returned and *errno* is set.